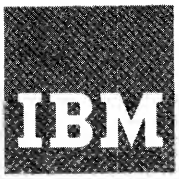
File No. 1410/7010-25
Form C28-0328-3



Systems Reference Library

IBM 1410/7010 Operating System (1410-PR-155)

FORTRAN–1410-FO-970

The IBM 1410/7010 FORTRAN language with its associated processor in the 1410/7010 Operating System provides the user with a convenient method of producing programs that will perform efficient scientific computation and data handling. FORTRAN source programs are written in a language similar to mathematics. The processor (1410-FO-970) compiles the source program directly into machine language in relocatable format.

This publication describes the types of arithmetic, control, input/output, subprogram, and specification statements accepted by the processor and the programming rules for their use.

MAJOR REVISION (December 1965)

This publication is a major revision of, and obsoletes, the publication *IBM 1410/7010 Operating System; FORTRAN*, Form C28-0328-2. The revision includes two diagrams of the control card setup needed to execute FORTRAN source and object programs, and a procedure for establishing an EQUIVALENCE that can handle double-subscripted variables. Changes to the text are indicated by a vertical line at the left of the affected text; changes to figures are indicated by a bullet (•) preceding the figure captions.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Programming Systems Publications, Department 637, Neighborhood Road, Kingston, New York 12401

Contents

Purpose of This Publication

This publication is a reference manual for persons writing programs in the FORTRAN language for use with the IBM 1410/7010 Operating System. Requirements for writing Autocoder subprograms to be combined with a FORTRAN program also are outlined.

FORTRAN Language and Processor

The 1410/7010 FORTRAN Programming System consists of a language and its associated processor. The FORTRAN language provides facilities for expressing most problems of numeric computation. In particular, problems containing large sets of formulas and many variables can be dealt with easily, and any variable may have up to three independent subscripts.

The capability of FORTRAN may be expanded by the use of subprograms. These subprograms may be written in FORTRAN language or in Autocoder, and may be called by other FORTRAN main programs or subprograms.

The language consists of five general categories of statements:

Arithmetic Statements define calculations to be performed.

Control Statements determine the processing flow.

Input and Output Statements specify the transfer of information between the computer and input/output devices.

Subprogram Statements allow the user to write subprograms.

Specification Statements declare properties of names appearing in the program and enable the user to control the allocation of core storage.

Any of these statements may be assigned a *statement number.* To permit reference within one statement to another statement, the latter statement must be assigned a statement number.

The 1410/7010 FORTRAN processor operates as part of the 1410/7010 Operating System. The object programs it produces are run according to the conventions of the Operating System.

Processor input is a source program written in the FORTRAN language. The processor lists the source program and produces an object program on cards, in card-image form on magnetic tape, and/or on disk storage. The object program is in the relocatable format of the 1410/7010 Operating System.

INCLUSION OF LIBRARY SUBROUTINES

Subroutines used to evaluate functions can be placed into the System Library where they are available for incorporation into object programs.

PROVISION FOR INPUT AND OUTPUT

Certain statements in the FORTRAN language introduce input and output routines into the object program. These routines permit considerable freedom of format in input and output data. The routines form a part of the System Library.

Prerequisite Publications

It is assumed that the user is familiar with the information contained in *IBM 1410/7010 Operating System; Basic Concepts,* Form C28-0318, and *General Information Manual; FORTRAN,* Form F28-8074.

Additional knowledge is necessary if it is desired to write subprograms in Autocoder as explained in Part 2 of this manual. This information appears in the following publications:

IBM 1410/7010 Operating System; Autocoder, Form C28-0326,

IBM 1410 Principles of Operation, Form A22-0526,

IBM 7010 Principles of Operation, Form A22-6726.

Definitions in Relation to the Operating System

The FORTRAN language defines *main program* and three classes of *subprograms* as explained in Part 1 of this publication. The FORTRAN term *main program* is synonymous with the Operating System term *primary subprogram,* and the FORTRAN term *subprogram* is synonymous with the Operating System term *secondary subprogram.* Main programs and subprograms may be separately compiled and both may call other subprograms. During execution of the object deck, all programming that constitutes the main program and any required subprograms resides in core storage at the same time.

The word *program* is used in this publication in a general sense where the distinction between a main program and a subprogram is not significant.

Minimum Machine Requirements

The minimum machine configuration required by the FORTRAN processor is discussed in the publication *IBM 1410/7010 Operating System: System Generation,* Form C28-0352.

Use and Contents of This Publication

This publication is divided into three parts, three appendixes, and an index:

Part 1 describes, with examples of use, the statements of the FORTRAN language.

Part 2 describes the operand parameters of the Monitor control card that institutes execution of a FORTRAN compilation, the format of the source program listing, calculation of the subscript expressions in a FORTRAN program, and basic requirements for writing Autocoder subprograms to be combined with a FORTRAN program.

Part 3 is a listing of the diagnostic and error messages.

Appendix A describes the order in which source program statements of a FORTRAN program are executed. Appendix B discusses preparing, checking, and punching a source program. Appendix C is a tabulation of the FORTRAN character set.

Constants, Variables, Subscripts, and Expressions

This section describes constants, variables, and subscripts for one-, two-, and three-dimensional arrays of variables. Also described are expressions which are combinations of constants, variables, and function references. (Functions and function references are discussed under "Subprograms: Function and Subroutine Statements.")

The 1410/7010 FORTRAN processor permits the user to define the precision of arithmetic calculations by specifying word size up to 20 digits for integers and 18 digits plus two exponent digits for real numbers. Specification is made on the Monitor control card that causes execution of the FORTRAN compiler. Details appear in Part 2 under "Monitor Card to Execute FORTRAN."

The word size specifications apply to both constants and variables of a given type (integers or real numbers). Word sizes for each type of number must be the same for all subprograms combined into a single program.

Constants

Two types of constants are permitted in a FORTRAN source program: integer constants and real constants. (In older material on FORTRAN these are referred to as fixed-point constants and floating-point constants, respectively.)

Integer Constants

General Form
An integer constant consists of n decimal digits, where $1 \leqq n \leqq k$, written without a decimal point. A preceding + or − sign is optional.

EXAMPLES

```
3
+1
−234567890
```

MAGNITUDE OF INTEGER CONSTANTS — THE VALUE OF k

The magnitude of an integer constant must be between 1 and (10^k-1) or be zero.

The k specification essentially defines core-storage *word* size for integer constants, since each integer constant in core storage will occupy k core-storage positions, which will be handled as a single unit (or *word*). For example, the constant +314 is stored (assuming k=5) as 00314. The constant −314 is stored as 00314. If the user attempts to use an integer constant of more digits than defined by k, the high-order digits are lost.

The value of k is indicated to the processor through control information supplied by the user. If k is specified by the user, the value of k must be $3 \leqq k \leqq 20$. If k is not specified by the user, the processor will use k equal to five decimal digits.

The constant zero is always stored with a positive sign.

Real Constants

General Form
A real constant consists of n decimal digits, where $1 \leqq n \leqq f$, written with a decimal point. A preceding + or − sign is optional. A real constant may be followed by a decimal exponent, which is written as the letter E followed by a (signed or unsigned) one- or two-digit integer constant.

EXAMPLES

17.		
5.0		
−.0003		
5.E3	i.e.	$5.0\text{x}10^3$
5.0E+3	i.e.	$5.0\text{x}10^{+3}$
5.0E−3	i.e.	$5.0\text{x}10^{-3}$
−5.0E+3	i.e.	$-5.0\text{x}10^{+3}$
−5.0E−3	i.e.	$-5.0\text{x}10^{-3}$
5.0E+03	i.e.	$5.0\text{x}10^{+3}$

MAGNITUDE OF REAL CONSTANTS — THE VALUE OF f

The magnitude of a real constant must lie between 10^{-100} and $(1-10^{-f}) \times 10^{99}$ or be zero.

The f specification for real number precision essentially defines core-storage word size for real constants. Within core storage a real constant is stored in an exponential form occupying f+2 digits (f+2 core-storage positions). The first f digits contain the fraction (a decimal point is understood to precede the

high-order digit position). The last two positions hold the exponent. Thus if f is 8, a real constant occupies ten core-storage positions – eight for the fraction and two for the exponent. For example, the constant +3.14159 is stored (assuming f=8) as 3141590001. The constant −3.14159 is stored as 3141590001.

The value of f is specified by means of control information supplied by the user to the processor. If f is specified by the user, its value must be $3 \leqq f \leqq 18$. If f is not specified by the user, the processor will use f equal to eight decimal digits. If the user attempts to use a real constant with more digits than defined by f the low-order digits are truncated.

Variables

A variable quantity is represented by a symbolic name. A variable is specified by its name and type. The type of variable (real or integer) corresponds to the type of constant (real or integer) that the values of the variable will assume.

Names of Variables

General Form
The name of a variable consists of one to six alphamcric characters, the first of which must be alphabetic. Within the same program, the same name must not be assigned to a variable end to a subprogram.

EXAMPLES

A
JOB5
COST
B546T

Types of Variables

The type of a variable, integer or real, can be specified in two ways: explicitly or implicitly.

EXPLICIT TYPE SPECIFICATION

Explicit type specification is made by the Type statements INTEGER and REAL. (See "The Specification Statements.")

IMPLICIT TYPE SPECIFICATION

Implicit type specification of a variable is made as follows:

If the first character of the variable name is I, J, K, L, M, or N, the variable is an integer variable.

If the first character of the name is *not* I, J, K, L, M, or N, the variable is a real variable.

Explicit type specification overrides implicit type specification. For example, if a variable name is INT and a type specification states that this variable is to be real, the variable is handled as a real variable even though it implicitly has the form of an integer variable.

Subscripts

A variable may be made to represent any element of a one-, two-, or three-dimensional array by appending one, two, or three subscripts, respectively to the variable name.

Form of Subscripts

General Form
Subscripts may take only the following forms: v c v+c v represents an unsigned, nonsubscripted integer variable. v−c c and c′ represent unsigned integer constants. c*v (+ denotes addition; −, subtraction; *, multiplication.) c*v+c′ c*v−c′

EXAMPLES OF SUBSCRIPTS

I
3
MU+2
MU−2
5*J
5*J+2
5*J−2

A variable in a subscript can not itself be subscripted.

Subscripted Variables

General Form
A subscripted variable consists of a variable name followed by a pair of parentheses enclosing one, two, or three subscripts separated by commas.

EXAMPLES

A(I)
K(3)
ALPHA(I, J+2)
BETA(5*J−2, K−2, L+3)

A reference to an array in a program must be preceded by a DIMENSION statement or a COMMON statement that specifies the size of the array. See the section "The Specification Statements" for the description of those statements.

Arrangement of Arrays in Core Storage

Arrays are placed in core storage in column order, in order of decreasing storage addresses:

One-Dimensional Arrays are stored sequentially.

Two-Dimensional Arrays are stored sequentially by column.

Three-Dimensional Arrays are stored sequentially by column from plane to plane. (That is, the first subscript is cycled most rapidly and the last least rapidly.)

For example, the array whose last element is A(M,N) will appear in core storage as:

A(M,N)..A(M,2)..A(2,2),A(1,2),A(M,1)..A(2,1),A(1,1)

where A(1,1) which was stored first, is in the high core-storage position, and A(M,N) is in the low position.

Expressions

Arithmetic Expressions

An arithmetic expression is a sequence of constants, subscripted or nonsubscripted variables, and function names (see "Subprograms: Function and Subroutine Statements"), separated by arithmetic operators, commas, and parentheses.

ARITHMETIC OPERATORS

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

RULES FOR CONSTRUCTION OF ARITHMETIC EXPRESSIONS

Figures 1 and 2 indicate which constants and variables may be combined by the arithmetic operators to form arithmetic expressions. Figure 1 gives the valid combinations for the arithmetic operators: + − * and /. Figure 2 gives the valid combinations for the arithmetic operator **.

+ - * /	Real	Integer
Real	Valid	Invalid
Integer	Invalid	Valid

Figure 1. Arithmetic Operators

		Exponent	
**		Real	Integer
Base	Real	Valid	Valid
	Integer	Invalid	Valid

Figure 2. Exponentiation

EXAMPLES

A+B	(Valid)
I*J	(Valid)
A*I	(Invalid)
J*B	(Invalid)
A+2	(Invalid)
A+2.	(Valid)
A+2.0	(Valid)
A**2.0	(Valid)
A**I	(Valid)
I**A	(Invalid)
I**2	(Valid)

Assume that A and B are of type real and I and J are of type integer.

Expressions may be connected by arithmetic operators to form compound expressions provided that no two operators appear in sequence and no operation symbol is assumed. For example, the algebraic expression

$(A \times B)(-C^D)$ must be written
(A*B)*(−C**D), *not*
(A*B)*−C**D or (AB)*(−C**D)

Parentheses may be used, as in algebra, to group expressions, to indicate hierarchy of operations, and to make interpretation easier for the user.

The mode of an arithmetic expression is either real or integer and, with the following exceptions, cannot be mixed:

A Real Quantity can appear in an integer expression as the argument of a function (see "Subprograms: Function and Subroutine Statements").

An Integer Quantity can appear in a real expression as the argument of a function, as a subscript, or as an exponent.

The expression A**B**C is not allowed. It must be written as (A**B)**C or A**(B**C), whichever is meant.

HIERARCHY OF OPERATIONS

Parentheses may be used, as in ordinary algebra, with any expression to specify the order in which operations are to be executed. When parentheses are omitted, the order of computation is the following:

1. Function computation and substitution (see "Subprograms: Function and Subroutine Statements")
2. Exponentiation
3. Multiplication and division
4. Addition and subtraction

When several operations within the same hierarchy appear in an expression, such as a sequence of consecutive multiplications and divisions (or additions and subtractions), the order of processing is from left to right, as illustrated in the following examples:

A+B+C is handled as (A+B)+C
I*J*K is handled as (I*J)*K
A+B−C is handled as (A+B)−C

I*J/K is handled as (I*J)/K
I*J*K/L is handled as ((I*J)*K)/L
A/B+C*D/E*F**G is handled as (A/B)+
(((C*D)/E)*(F**G))

EXPONENTIAL EXPRESSION SIGN RESTRICTIONS

The following restrictions on the signs of the base and exponent must be observed. If overflow occurs during object program execution, it is indicated as shown in the third column.

FORM OF EXPONENTIAL EXPRESSION	RESTRICTION ON SIGN OF BASE OR EXPONENT	RESULT IF ANSWER CREATES OVERFLOW
A**B	Negative base (A) not permitted; error message produced by the ALOG function and unusual end of program occurs. When base (A) is zero, result is zero for any value of B. A**B is computed from EXP (B*ALOG (A)). The ALOG function does not accept negative values of the argument. In applications where negative A is expected and B is identically integral (e.g., 2.0 or −6.0), use A**IFIX(B) instead of A**B. Refer to "Built-In Function" for an explanation of EXP, ALOG, and IFIX functions.	Unusual end of program and error message.
A**I	When base (A) is zero, result is zero for any value of I.	Overflow indicator turns on and exponent is 99.
I**J	Negative exponent (J) always results in an integer 1 and an error message. When base (I) is zero, result is zero for any value of J.	High-order digits of the integer result are truncated.

USE OF EXPONENTIAL AND EXPANDED FORMS

If use of the exponential forms A**I or I**J and the expanded forms A*A*. . . or I*I*. . . is optional, an improvement in program efficiency can be obtained by selection of the form shown here.

INTEGER EXPONENT	MOST EFFICIENT FORM
0-5	Expanded form; e.g., A*A and A*A*A
6 or greater	Exponential form; e.g., A**I

Relational Expressions

A relational expression consists of two arithmetic expressions, of the same mode, separated by a relational operator.

RELATIONAL OPERATORS

.GT. Greater than (>)
.GE. Greater than or equal to (≧)
.LT. Less than (<)
.LE. Less than or equal to (≦)
.EQ. Equal to (=)
.NE. Not equal to (≠)

NOTE: The preceding and following periods are a necessary part of the symbol.

RULES FOR CONSTRUCTING RELATIONAL EXPRESSIONS

Figure 3 indicates the valid combinations for the relational expression a ϕ b, where a and b are arithmetic expressions, and ϕ is any relational operator.

	ϕ	b: Real	b: Integer
a	Real	Valid	Invalid
a	Integer	Invalid	Valid

Figure 3. Relational Operators

EXAMPLES

A.GT.B (Valid)
10..LE.A (Valid)
I.EQ.J (Valid)
A**2..NE..01 (Valid)
A.GE.I (Invalid)

Assume that A and B are of type real and I and J are of type integer.

NOTE: The arithmetic expressions can contain function references (see "Subprograms: Function and Subroutine Statements").

HIERARCHY OF OPERATIONS

A relational expression is computed in the following way: the value of each arithmetic expression is computed following the rules of hierarchy for arithmetic expressions; these values are then compared for the relation indicated by the relational operator.

USE OF RELATIONAL EXPRESSIONS

Relational expressions are used only in the Relational IF statement (see "The Control Statements").

The arithmetic statement defines a numeric calculation. A FORTRAN arithmetic statement closely resembles a conventional arithmetic or algebraic formula, with the primary difference that the equal sign (=) specifies replacement, rather than equality.

General Form
a=b a is a real or integer variable that may or may not be subscripted. b is an arithmetic expression.

EXAMPLES

```
A=B + (C - 1.0)**D
I=J -- K/(L + 1)
A(I,J,K) = D(N) + DAV**MAR
```

A real or integer arithmetic expression can be equated to any type of variable.

If the type of variable is *real* and the mode of the expression is *integer*, the expression is evaluated and this value is converted to a real value. The reverse is also valid. These and additional examples of arithmetic statements are:

I = B	Truncate B to an integer and convert it to an integer value; store it in I.
A = I	Convert I to a real value and store it in A.
A = B	Store the value of B in A.
I = I + 1	Add 1 to I and store in I. This example illustrates that an arithmetic statement is not an equation. Rather, it is a command to replace a value.
A = 3.0*B	Multiply 3 by B and store result in A.

The Control Statements

Control statements enable the user to control the flow of his program.

Unconditional GO TO Statement

General Form
GO TO n n is a statement number.

This statement causes control to be transferred to the statement numbered n. (See Appendix B for a discussion of statement numbers.)

EXAMPLE

```
GO TO 57
```

Computed GO TO Statement

General Form
GO TO (n_1, n_2, . . . ,n_m),i n_1, n_2, . . . ,n_m are statement numbers. i is a nonsubscripted integer variable. The limits of the value of i are: $1 \leqq i \leqq m$.

This statement causes control to be transferred to statement number n_1,n_2, . . . , or n_m, depending on whether the value of i at the time of execution of the statement is 1, 2, . . . , or m, respectively.

EXAMPLE

```
GO TO (30, 40, 50, 9), K
```

Thus, if the value of K is 3 at the time of execution of this statement, the program will transfer to the statement identified by the third statement number in the list, statement 50.

Relational IF Statement

General Form
IF (relational expression) statement The statement may be any executable FORTRAN statement except another Relational IF statement or a DO statement.

The Relational IF statement will cause the statement following the parenthesis to be executed if the relational expression is true. If the relational expression is not true, control will transfer to the next sequential statement in the program.

If the relational expression is true and the statement is an arithmetic statement (e. g., A = B*C), the arithmetic operations are performed and control is then transferred to the next sequential statement.

If the relational expression is true, and the statement is a CALL (see "Subprograms: Function and Subroutine Statements"), control will be transferred to the next sequential statement upon return from the subprogram called.

EXAMPLES

```
IF (L.GE.16) ANSWER=A/B-C
IF (A.GE.0.0) GO TO 876
```

Arithmetic IF Statement

General Form
IF (arithmetic expression) n_1, n_2, n_3 n_1, n_2, n_3 are statement numbers.

Control is transferred to statement number n_1, n_2, or n_3 depending on whether the value of the expression is less than, equal to, or greater than zero, respectively.

EXAMPLES

```
IF (A) 2, 3, 2
IF (A-B) 10, 5, 7
IF (IOTA-KAPPA) 1, 2, 3
```

DO Statement

General Form
DO n i = m_1,m_2,m_3 n is a statement number (see Appendix B). i is a nonsubscripted integer variable. m_1,m_2,m_3 are each either unsigned integer constants or nonsubscripted integer variables. m_3 is optional; if it is not stated, its value is assumed to be 1. If it is omitted, the preceding comma must also be omitted.

EXAMPLES

```
DO 30 I=1, M, 2
DO 24 I=2, 10
```

The DO statement is a command to execute repeatedly the statements that follow, up to and including the statement numbered n. The first time the statements are executed, i has the value m_1 and each succeeding time i is increased by the value of m_3. After the statements have been executed with i equal to the highest value that does not exceed m_2, control passes to the statement following statement number n. This is called a normal exit from the DO statement.

The Range of the DO Statement: The range of the DO statement is the set of statements that will be executed repeatedly.

The Index of the DO Statement: The index of the DO statement is the variable i. Its value is available during execution of the DO. After a normal exit from a DO, the value of the index is not available for use.

DO's Within DO's: A DO can be contained within another DO: this is called a nest of DO's. If the range of a DO contains another DO, then all statements in the range of the enclosed DO must be within the range of the enclosing DO. The maximum depth of nesting, including implied DO's in I/O lists, is 25. That is, a DO can contain a second DO; the second can contain a third; the third, a fourth; and so on up to 25 DO statements.

Transfer of Control: Control may not be transferred into the range of a DO from outside its range. However, control can be transferred out of a DO range. In this case, the value of the index remains available for use. If exit is caused by transfer out of the ranges of a set of nested DO's, then the index of each DO is available.

Figure 4 illustrates the possible transfers in and out of the range of a DO. In this figure, 1, 2 and 3 are permitted, but 4, 5, and 6 are not permitted.

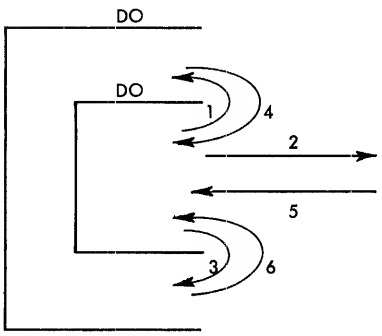


Figure 4. Transfer of Control, DO Statements

Restrictions: Any statement that redefines the value of the index or any of the indexing parameters (m's) is not permitted in the range of a DO. When the range of a DO contains reference to a subprogram, care must be taken that the subprogram does not alter the index or any of the indexing parameters. The range of the DO cannot end with a GO TO statement (see the "CONTINUE Statement," below).

CONTINUE Statement

General Form
CONTINUE

CONTINUE is a dummy statement that does not produce any executable instructions. It is used as the last statement of a DO to provide a branch address for GO TO statements that are intended to begin another repetition of the DO range. An example is:

```
      .
      .
      .
   DO 10 I = 1, 20
      .
      .
11 X = Y + Z
      .
      .
      .
      .
   IF (A.GE.B) GO TO 10
   A = A + 1.0
   B = B - 2.0
   GO TO 11
10 CONTINUE
      .
      .
      .
```

PAUSE Statement

General Form
PAUSE PAUSE n n is an unsigned integer constant whose value is less than 10^5.

This statement causes the program to print on the console printer "PAUSE 00000" or, if n is specified, "PAUSE n" (where n includes leading zeros). The program will then enter a waiting loop. Operator intervention will cause the program to resume execution, starting at the next statement after the PAUSE statement.

STOP Statement

General Form
STOP

This statement terminates the execution of the program and returns control to the Monitor.

END Statement

General Form
END

The END statement defines the end of a program or subprogram for the processor. Physically, it must be the last statement of each program or subprogram. The END statement is not executable; it must not be encountered in the flow of the program.

RETURN Statement

General Form
RETURN

This is the normal exit from a subprogram. The RETURN statement signifies a logical conclusion of the computation and returns any value computed and control to the calling program.

Input/Output Statements

The Input/Output (I/O) statements control the transmission of information between the computer and input/output devices (such as card readers, card punches, and magnetic tape units). The I/O statements fall into the following general categories:

FORMAT Statements: These are nonexecutable statements that specify (a) the arrangement of the information to be transferred, and (b) the editing transformation between internal (core-storage) and external forms of the information. The FORMAT statements are used in conjunction with the general I/O statements.

General I/O Statements: These statements cause transmission of information between the computer and input/output devices. They are READ and WRITE.

Manipulative I/O Statements: These statements manipulate input/ouput units. They are END FILE, REWIND, and BACKSPACE.

Specification Lists

The general I/O statements call for the transmission of information and must, therefore, include a list of the items to be transmitted. A list item may be a subscripted or nonsubscripted variable; successive items of the list must be separated by commas. An I/O list is read from left to right. A constant may appear in the list only as a subscript or as an indexing parameter.

A list is ordered, and its order must be the same as the order in which the information appears in the input medium or in which it is desired that the information appear in the output medium.

A list may contain implied DO's. In this case a comma must precede the index variable, and all items to be included in the range of the implied DO must be set off by parentheses. DO's can be effectively nested by the placing of matching parentheses around the first and last items of each successive inner DO range.

The index values for the implied DO may appear in the list. For example, the list specification

K, L, M, (E(I, J), I=K, L, M)

will transfer the values of the integer variables K, L, and M, and will also insert those values into the implied DO. For example, if K=1, L=99, and M=5, the implied DO is effectively

E(I, J), I=1, 99, 5

EXAMPLE

For the following example, assume that the value of K is defined in the program before the appearance of the general I/O statement of which the list is a part. Consider this list:

```
A, B(3), (C(I), D(I, K), I=1, 10, 2),
((E(I, J), I=1, 10, 2), F(J, 3), J=1, K)
```

If this list is used with an output statement, the information will be written on the output medium in this order:

```
A, B(3), C(1), D(1, K), C(3), D(3, K),
                . . . , C(9), D(9, K),
E(1,1), E(3,1), . . . , E(9, 1), F(1, 3),
E(1,2), E(3,2), . . . , E(9, 2), F(2, 3),
. . . . . .
. . . . . .
E(1,K),E(3,K), . . . , E(9, K), F(K, 3)
```

Similarly, if this list were used with an input statement, the variable names A, B(3), C(1), etc., will be assigned to the values given on the external medium.

IMPLIED DO

The order of the list generated by the implied DO's of the preceding example is approximately equivalent to the following sequence of statements. The order is *approximately* equivalent since each READ statement below implies that input is to come from the beginning of a new input record. This may not be the case for the actual list.

```
  READ (M, n1) A
  READ (M, n2) B(3)
  DO 5 I=1, 10
  READ (M, n3) C(I)
5 READ (M, n4) D(I, K)
  DO 9 J=1, K
  DO 8 I=1, 10, 2
8 READ (M, n5) E(I, J)
9 READ (M, n6) F(J, 3)
```

n_n are the statement numbers of FORMAT statements, as explained further on in this section.

M is the symbolic unit designated for input.

Reading or Writing Entire Arrays

When the reading or writing of an entire array is required, an abbreviated notation may be used in the list of the input or output statement. Only the name of the array need be given, and subscripts may be omitted. For example, if A has previously been listed with a DIMENSION or a COMMON (With Dimensions) statement (see "The Specifications Statements"), the statement

READ (1, 18) A or READ (4) A

will cause all of the elements of the A array to be read in the implied order of elements. Thus, if A is a 2×3 array, the elements should be placed on the input medium in this order:

A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3).

FORMAT Statement

General Form
FORMAT (S_1, S_2, . . . , S_n) S_1, S_2, . . . , S_n are format specifications.

EXAMPLE

FORMAT (I2, E12.4, F10.2)

The general formatted input/output statements — READ (i, n) List, and WRITE (i, n) List — in addition to a list of items to be transmitted, refer to a FORMAT statement that describes the data record and the type of conversion to be performed between the internal and the external representations for each element in the list. The FORMAT statement describes the record to be read or written by giving the specification for each field — numeric, alphameric, or blanks — in the record from left to right, beginning with the first character of the record.

FORMAT statements must appear in the source deck after any EQUIVALENCE statements and before any Statement Functions or executable statements. FORMAT statements must have a statement number. See Appendix B for a discussion of statement numbers.

A total of 9,999 characters is permitted in the FORMAT statements of each program compiled. Blanks are not counted except when specified in the H conversion described later.

Format Specifications

Numeric Fields

Three types of specifications are available for information in numeric form:

INTERNAL REPRESENTATION	CONVERSION CODE	EXTERNAL REPRESENTATION
Real (e.g., 1230000006 for f=8)	E	Real with exponent (e.g., .123E−06)
Real (e.g., 1230000006 for f=8)	F	Real without exponent (e.g., .000000123)
Integer (e.g., 00123 for k=5)	I	Integer (e.g., 123)

These types of conversion are specified in the following forms:

Ew.d

Fw.d

Iw

E, F, and I specify the type of conversion.

w is an unsigned integer constant specifying the field width of the data; w must not be zero. (This specification may be greater than that required for the actual digits, to provide spacing between successive numbers.)

d is an unsigned integer constant specifying the number of positions of the field that are to appear as a fractional part.

Specifications for successive fields within a record are separated by commas. Specification of more characters than are permitted for the appropriate input/output record cannot be given. For example, the format specification for output on a printer should not provide for more characters than can be handled by the printer.

Information to be transferred under E and F conversion must be of type real; information to be transferred under I conversion must be of type integer.

In E and F type FORMAT statements, w may consist of up to three numerical characters (maximum value, 133; see message 097) and d of up to two numerical characters.

However, IBCOMMON does not handle more than two numerical characters for both w and d; for this reason fields w and d should never exceed two numerical characters.

OUTPUT FIELDS

The field-width count (w) for output E and F conversion must include a space for the decimal point; a space for the sign must be included only if negative numbers are to be converted. (An example of the input and output form of negative numbers subject to E or F conversion is −.123Eb06, −12.3, etc., where b indicates a blank.) Also, for E conversion, a count must be made for the letter E, the sign of the exponent, and the two-digit exponent. Therefore, for E output conversion, minimum w = d + 6.

Two examples of E output conversion follow:

INTERNAL REPRESENTATION	FORMAT SPECIFICATION	OUTPUT REPRESENTATION
1230000006	E9.3	b. 123E−06
1230000006	E9.3	−. 123Eb06

Nonsignificant zeros do not appear in the output for E, F, or I conversion, except for the exponent of E output conversion. If the exponent is less than ten, a zero precedes the significant exponent digit.

If a number converted by E or F output conversion requires more space than is allotted by the format specification, an asterisk (*) is inserted in the high-order position of the field, any other digits preceding the decimal point are replaced by blanks, the decimal point is inserted in the correct position, and digits following the decimal point are replaced by zeros. For E conversion the internal exponent digits are inserted in the external exponent field. The external exponent digits, therefore, are correct unless overflow occurred within the machine.

If a number converted by output I conversion requires more space than allotted by field width in the format specification, the excess high-order digits will be lost, an asterisk (*) will be inserted in the leftmost

position of the field, and an error message will be supplied during execution.

If the number requires fewer spaces than allotted, the high-order excess positions are filled with blanks. Thus, a format specification that has a greater field width than is required may be used to space an output record.

INPUT FIELDS

The field-width count (w) for input E and F conversion must include a space for a decimal point if one is used explicitly in the input data (see "Data Input to an Object Program"). A space for the sign must be included only if negative numbers are to be converted.

For E input conversion a variety of forms, including the standard output form, is acceptable. A count must be made for each character of the input data that may be present. This includes the E, the sign of the exponent, and the exponent digits.

The following example shows the forms of input data acceptable for E conversion.

VALUE TO BE REPRESENTED	PERMISSIBLE INPUT FORM UNDER FORMAT (E10.3)
$.123 \times 10^{-6}$	+0.123E−06 +.123E−06 .123E−06 .123E−6 .123−06 .123−6 Decimal point in any of above forms may be omitted.
$-.123 \times 10^{6}$	−0.123E+06 −.123E+06 −.123E+6 −.123Eb06 −.123E6 −.123Eb6 −.123+06 −.123+6 Decimal point in any of above forms may be omitted.

Equivalent forms of the same number, such as 1.23−7 or 12.3E−8 for the first number in the example, also are acceptable.

If the FORMAT specification does not describe the input data correctly, the desired transfer cannot occur. For example, if −1234.5 is to be read into core storage under F5.2, the number is treated as though it were −12.34.

Alphameric Fields

FORTRAN provides two methods by which alphameric information may be transferred:

The specification Aw causes w characters to be read into or written from a core-storage location designated by a variable or array name.

The specification nH specifies that alphameric information is contained in the FORMAT statement.

The basic difference between A and H conversion is that alphameric information handled by A conversion is given a name, and thus can be referred to by this name for processing and modification. The associated I/O statement therefore requires a list when A conversion is specified by the FORMAT statement.

Information handled by H conversion is not labeled; it is a constant field and cannot be referred to or manipulated.

For input, the specification nAw causes n successive fields of w characters each to be read in the form in which they appear in the input medium. The n names specified by the list are assigned to the n fields that are read into core storage.

A CONVERSION

For example:

```
      .
      .
      .
8 FORMAT (12A6)
      .
      .
      .
READ (1, 8) X, Y, (ACONV (I), I=1, 10)
      .
      .
      .
```

These statements cause the Standard Input Unit to read a card containing 12 six-character fields into core storage in the form that the fields appear on the card. The first word is assigned the name X; the second Y; and the remaining words are ACONV(1) *through* ACONV(10).

For output, the specification nAw causes n successive fields of w characters each to be transferred from core storage to the device specified.

In both input and output, w must not exceed the word size of the list elements. That is, w must not be greater than k (for integer elements in the list) or f+2 (for real elements in the list).

With A conversion, input is read into core storage in inverted order. If this data is printed from core storage with any other conversion, the output appears in its inverted order. For example, with A conversion, VELOCITY would be read into core storage as YTICOLEV. If A conversion is used for output, it would be printed as VELOCITY. However, if any other conversion is used for output, this data would be printed as YTICOLEV.

H CONVERSION

The specification nH is followed by n alphameric characters in a FORMAT statement. A comma must separate successive specifications, including the H conversion, used in the FORMAT statement. The separating comma must appear after the last alphameric character; the last character may be a blank. An example is . . . , 4HABCb, . . .

For input, n characters are extracted from the input record and replace n characters of the appropriate source program FORMAT statement.

For output, the n characters following the specification (or the characters that replace them through the action described above) are written as part of the output record. If the WRITE statement refers to the Standard Print Unit and the first specification for a record is an H conversion, the first character of the alphameric information is not printed but is used to control vertical spacing of the carriage of the printer.

For example:

```
      .
      .
      .
9 FORMAT(26HbbbTHISbISbALPHAMERICbINFO)
      .
      .
  WRITE (3, 9)
      .
      .
      .
```

These statements cause the specified heading to be printed, indented two spaces from the left, by the Standard Print Unit. Note that blanks are considered in A and H conversion to be alphameric characters and must be included as part of the character count.

Blank Fields, X Conversion

The specification nX introduces n blank characters into an input/output record. The number n must always be less than 133 (the maximum record size). A comma must separate successive specifications, including the X conversion, used in the FORMAT statement.

For input, nX causes n characters of an input record to be ignored.

For output, nX causes n blank characters to be placed into the output record. This conversion is used to space within an output record.

Repetition of Field Format

It may be desired to transfer n successive fields within the same record with the same format specifications. This is indicated by placing a number n (an unsigned integer constant) before the E, F, I, or A. Thus, the specification 3E12.4 is the equivalent of E12.4, E12.4, E12.4.

Repetition of Groups of Fields

Limited parenthetical expressions are permitted in format specifications to indicate the repetition of data fields within a record. One pair of nested parentheses, in addition to the parentheses required by the FORMAT statement, is permitted. For example:

```
FORMAT (2(F10.6, 3E12.2), I6) is valid, but
FORMAT (2(F10.6, 3(E12.2, I6) ) ) is not valid.
```

The valid FORMAT example above is equivalent to

```
FORMAT (F10.6, E12.2, E12.2, E12.2, F10.6, E12.2,
.E12.2, E12.2, I6).
```

Scale Factors, P Conversion

To permit a general use of E and F conversion, a scale factor followed by the letter P may precede a specification.

The scale factor is defined for F input conversion as follows:

$10^{-(\text{scale factor})} \times \text{external quantity} = \text{internal quantity}$

The scale factor is defined for E and F output conversion as follows:

$\text{external quantity} = \text{internal quantity} \times 10^{(\text{scale factor})}$

For input, P conversion can be used only with F conversion. For example, if input data is in the form xx.xxx, and it is desired to use it internally in the form .xxxxx, the specification that will make this change is 2PF7.3.

For output, P conversion may be used with both E and F conversions. The following examples of F conversion use the same data; vertical line separate the four adjacent fields.

Specification	Data Fields			
I2, 3F11.3	27	bbbb-93.209	bbbbbb-.007	bbbbbbb.553
I2,-1P3F11.3	27	bbbbb-9.320	bbbbbb-.000	bbbbbbb.055
I2, 1P3F11.3	27	bbb-932.097	bbbbbb-.075	bbbbbb5.536

A positive scale factor used for output with E conversion increases the base and decreases the exponent. The following example shows this effect (using the same data as in the previous examples).

Specification	Data Fields			
I2, 1P3E12.4	27	b-9.3209Eb01	b-7.5804E-03	bb5.5362E-01
I2, 3E12.4	27	bb-.9320Eb02	bb-.7580E-02	bbb.5536Eb00

The scale factor is assumed to be zero if no value is given. However, once a value has been given, it will hold for all E and F conversions following the scale factor within the same FORMAT statement. Thus, the specification

```
1PE12.4, E14.5, F8.3
```

is equivalent to:

```
1PE12.4, 1PE14.5, 1PF8.3
```

If it is desired to have only the first item in that specification affected by P conversion, the specification should be written:

```
1PE12.4, 0PE14.5, F8.3
```

Multiple-Record FORMAT Statements

To deal with many records, a single FORMAT statement may have several single-record format specifications

separated by a slash (/) to indicate the beginning of a new record.

For example:

```
FORMAT (3F9.2, 2F10.4/8E14.5)
```

will transfer the first, third, fifth, ... records with the specification 3F9.2, 2F10.4; and the second, fourth, sixth, ... records with the specification 8E14.5.

If a single multiple-record FORMAT statement is required in which, for example, the first two items are unique and all the remaining items are to be transferred to the same specification, the specification for these remaining items must be enclosed in a pair of parentheses.

For example:

```
FORMAT (I2, 3E12.4/(10F12.4))
```

would transfer the first record with the specification I2, 3E12.4 and all succeeding records with the specification 10F12.4. That is, the repetition starts from the last left parenthesis.

If data items remain to be transferred after the specifications have been "used," the specification will repeat from the last left parenthesis.

For example, in the statement

```
FORMAT (I2, 4E12.4/(3F12.4))
```

the specification used for repetition is (3F12.4)

The equivalent of blank lines between output records, or records skipped for input records, may be introduced into a multiple-record format specification by consecutive slashes. The number of records skipped, or blank lines inserted, is a function of the number and placement of the slashes, as summarized in the following table.

FOR N CONSECUTIVE SLASHES	INPUT RECORDS SKIPPED OR BLANK LINES INSERTED IN OUTPUT
At the beginning of the format specifications — e.g., FORMAT (///I6)	n
In the middle of the format specifications — e.g., FORMAT (I6///I6, 19)	n − 1
At the end of the format specifications — e.g., FORMAT (I6///)	n

For example, if the statement FORMAT (I2, E12.4//// F12.3) is used for printed output, three blank lines will be inserted between the data specified by I2, E12.4 and the data specified by F12.3.

Carriage Control

If the "i" in a formatted WRITE statement refers to the Standard Print Unit, the first character in each output record is used to control the vertical spacing of the carriage of the printer for vertical forms control. Carriage control characters are listed in the publication *IBM 1410/7010 Operating System; System Monitor,* Form C28-0319. The character can be placed in the output record by means of A or H conversion. A blank causes normal single spacing before the line is printed. The carriage control character also can come from X, I, E, or F conversion. (Horizontal forms control is determined by FORMAT statement specifications.)

I/O List and FORMAT Statement Relationship

The list included in each general I/O statement designates the data to be transmitted from the input medium or to the output medium. However, the sequence of information within a record is controlled by the FORMAT statement. H and X conversions are read from, or are placed in, the record in the sequence indicated in the FORMAT statement. E, F, I, or A conversions, when specified in the FORMAT statement, operate upon the first unused item in the list.

Even though a FORMAT statement may handle more information fields than are indicated by items in the I/O list, execution of the I/O statement terminates when the last item on the list is transmitted and any immediately following FORMAT specifications not requiring a list element are completed.

If items remain to be transferred after the FORMAT specification is "used," the specification is reused until all items are transferred as described earlier under "Multiple-Record FORMAT Statements."

As an example of correspondence between items of the list and the FORMAT specification, consider the following two statements:

```
      .
      .
      .
      .
12 FORMAT (10X, 15HAPPRAISEDbVALUE//
                 (6HbWARDb, I2, 5X, F8.2))
      .
      .
WRITE (3, 12) (K, VALUE(K), K=1, J)
```

The purpose of the example is to print results of calculations made, by ward, of the average appraised value of residential units. If J = 14, the printed matter has this appearance:

```
          APPRAISED VALUE
WARD 1      12654.12
      .
      .
      .
WARD 14     26223.68
```

Since the FORMAT statement governs the sequence of information in a line, nine spaces are left and APPRAISED VALUE is printed. One line is skipped. On the

second line, WARD followed by a space appears first. This is followed by an integer which may be up to two digits. The integer to be inserted is the first item on the list, K. Five blank spaces are next inserted. The list is again referred to for the name associated with F8.2, VALUE (K). Successive lines are printed until the list is exhausted since each re-scan implies a new record.

Data Input to an Object Program

Data input to an object program is contained in records conforming to the specifications described below.

1. The maximum formatted record length is 133 characters.
2. The data must correspond in order, type, and field width to the specifications in FORMAT statements. Reading starts with the first character position.
3. Plus signs are indicated by either a blank (C bit, no punch) or a "+" (C,B,A bits, 12 punch). Minus signs are represented by a "−" (B bit, 11 punch).
4. Blanks within numeric fields are regarded as zeros.
5. Data for E and F conversion may contain any number of digits, but only the high-order f digits will be retained (see "Constants" in Part 1). Numbers for I conversion may contain any number of digits, but only the low-order k digits are retained.
6. As previously explained, numbers for E conversion need not have all columns devoted to the exponents; that is, Esdd (where s is the sign and dd the exponent) need not have a leading zero if it is less than 10. This and other valid forms are:

E+2, E2, +2, +02, Eb02, Eb2, E−22, E−2, and −2.

7. Numbers for E and F conversion need not have the decimal point punched in the card; the format specification will supply the required decimal point. For example, −09432+2, with the input specification E12.4, will be treated as if the decimal point is punched between the zero and the 9. (The 4 in the specification E12.4 will produce four decimal places.) If a decimal point is punched, it will override the position specified by the format specification.
8. If cards contain numbers for E conversion, the numbers must be punched in the low-order positions of their respective fields, w.

Symbolic Input/Output Unit Designation

Input and output units are referred to *symbolically* in I/O statements. These references are indicated as "i," an unsigned integer constant or integer variable in the descriptions of the general forms of these statements. The correspondence between the symbolic units and the actual physical devices is made when the object program is to be run. For a description of control information and procedures required, see the publication *IBM 1410/7010 Operating System; System Monitor*, Form C28-0319.

If i is a variable name, this name must be assigned a numeric value by the program before the I/O statement is executed. Any FORTRAN statement or operation that assigns a numeric value to the variable name may be used. The form of the constant i is restricted to a single integer digit as shown below; e.g., 04 is invalid.

At the time of execution of the I/O statement, the numeric value of i determines which Operating System unit is operated on as follows:

VALUE OF i	SYMBOLIC UNIT
1	Standard Input Unit
2	Standard Punch Unit
3	Standard Print Unit
4	MW1
5	MW2
6	MW3
7	MW4
8	MW5
9	MW6

(Work Tapes)

Only formatted input operations, i.e., READ (1,n), can be performed on unit 1; only formatted output operations, i.e., WRITE (2,n) and WRITE (3,n), can be performed on units 2 and 3; all I/O operations can be performed on the work tapes designated as units 4 through 9.

FORTRAN work tapes have the label characteristics used for the system files. FORTRAN uses the IOCS to write the standard labels, which include the number of characters in the label, the file serial number, file identification, the creation date, and the reel sequence number. Information concerning tape labels is presented in greater detail in *IBM 1410/7010 Operating System; Basic Input/Output Control System*, Form C28-0322.

General Input/Output Statements

Input — The READ Statement

There is one input statement: READ. This statement is used to transfer information from input devices to the computer.

General Forms
READ (i, n) List READ (i) List i is an unsigned integer constant or integer variable specifying the symbolic unit to be used for data input. n is the statement number of the FORMAT statement describing the data to be transferred. List is an input list.

EXAMPLES

```
READ(1, 3)A,(B(I),I=1,99)
READ(L)J, (B(I),I=J,99)
```

The READ(i, n)List statement causes information to be read from symbolic unit i according to FORMAT statement n.

The READ(i)List statement causes information in internal format, as on a work file, to be read from symbolic unit i into core storage. The information must have been previously written with the WRITE (i) List statement.

The first form of the READ statement reads in successive data records (or parts of a data record) until the entire list is satisfied; that is, until all data items specified by the list have been read, converted, and stored.

The list in the second form of the READ statement must not be longer than the number of words in a record. If the list is equal to the word count, the entire record is read. If the list is shorter than the word count, the unread items in the record are skipped. This form of the READ statement does not require format specifications, as no conversion is required.

Output — The WRITE Statement

There is one statement that is used to transfer information from the computer to output devices: WRITE.

General Forms
WRITE (i, n) List WRITE (i) List i is an unsigned integer constant or integer variable specifying the symbolic unit to be used for data output. n is a FORMAT statement number. List is an output list.

EXAMPLES

```
WRITE (J,3)A,(B(I),I=1,99)
WRITE (4)J,(B(I),I=J,99)
```

The first form of the WRITE statement causes information to be written on symbolic unit i according to FORMAT statement n. The information is recorded in one or more physical records as specified by the FORMAT statement.

The second form of the WRITE statement causes information to be written in internal format on symbolic unit i. A FORMAT statement is not used since no conversion is performed. The information specified by the list is considered to be one logical record although it may be written as more than one physical record.

The unformatted form can be used to write a scratch file for internal use by an object program.

Manipulative Input/Output Statements

The statements END FILE, REWIND, and BACKSPACE manipulate work tapes, units 4 through 9, as described below.

END FILE Statement

General Form
END FILE i i is an unsigned integer constant or integer variable specifying the symbolic unit.

The END FILE statement causes a tape mark to be written on symbolic unit i.

EXAMPLES

```
END FILE 5
END FILE N
```

REWIND Statement

General Form
REWIND i i is an unsigned integer constant or integer variable specifying the symbolic unit.

The REWIND statement causes the tape reel mounted on symbolic unit i to be rewound.

EXAMPLES

```
REWIND 4
REWIND N
```

BACKSPACE Statement

General Form
BACKSPACE i i is an unsigned integer constant or integer variable specifying the symbolic unit.

The BACKSPACE statement causes the tape reel mounted on symbolic unit i to be backspaced one physical record if the tape was written under FORMAT control, or one logical record (which may consist of more than one physical record) if the tape was written without FORMAT control. The logical record that is backspaced consists of the contents of the list of the associated WRITE (i) List statement.

EXAMPLES

```
BACKSPACE 9
BACKSPACE N
```

The FORTRAN language defines Statement Functions and three classes of subprograms: Built-In Functions, FUNCTION subprograms, and SUBROUTINE subprograms. Their uses and differences are discussed in this section of the manual.

Advantages of Subprograms

The advantage of subprograms stem primarily from their ability to be compiled separately. A program can be written as a short main program and a number of subprograms. Changes or error correction can then be made to the segmented program by the re-compiling of only the affected subprograms.

Other advantages are that any subprogram can be placed in the System Library for use with other programs and that program segmentation permits more than one person to be simultaneously writing a large program.

Functions and SUBROUTINE Statements

As a group, Statement Functions, Built-In Functions, and FUNCTION subprograms can be simply called functions. Functions differ from SUBROUTINE subprograms in that functions always return a single result to the calling program, whereas SUBROUTINE subprograms may return more than one value to a calling program.

Naming Subprograms and Statement Functions

Statement Functions and subprograms are named in the same manner as FORTRAN variables (see "Constants, Variables, Subscripts, and Expressions").

A subprogram name consists of one to six alphameric characters, the first of which must be alphabetic.

The type (real or integer) of a Statement Function may be indicated implicitly by the initial character of the name, or explicitly by a Type statement (see "The Specification Statements"). In the latter case the implicit type is overridden by the explicit specification.

The type (real or integer) of a Built-In Function is already specified (Figure 5) and need not be defined by the user.

The type (real or integer) of a FUNCTION subprogram may be indicated implicitly by the initial character of the name or explicitly by a Type statement. In the latter case the implicit type is overridden by the explicit specification.

The type (real or integer) of a SUBROUTINE subprogram is not defined since the result returned to the main program is dependent only on the type of the variable names in the argument list.

Definition and Usage of Subprograms — Valid Components

The following tables summarize the FORTRAN language components that are valid in the definition and usage of Statement Functions and subprograms.

Definition of Subprograms

The following table refers to the kinds of arguments listed in the "a" portion of the Statement Function general form and in the FUNCTION or SUBROUTINE statement.

As Arguments in the Definition	Statement Function	Built-In Function	FUNCTION Sub-program	SUB-ROUTINE Sub-program
Constant		Pre-defined		
Simple Variable	X		X	X
Subscripted Variable				
Array Name			X	X
Arithmetic Expression				
External Name[1]			X	X

[1]The name of a FUNCTION or SUBROUTINE subprogram cannot be the same as the name of an argument of that subprogram. For example, the following is *invalid:* FUNCTION DAV (I,D,DAV)

The following table refers to the kinds of arguments listed in the "b" portion of the Statement Function general form and the kinds of arguments that may be used in subprograms headed by the FUNCTION or SUBROUTINE statement.

In the Body of the Definition	Statement Function	Built-In Function	FUNCTION Sub-program	SUB-ROUTINE Sub-program
Constant	X	Pre-defined	X	X
Simple Variable	X		X	X
Subscripted Variable			X	X
Array Name			X	X
Arithmetic Expression	X		X	X
External Name	X		X	X

Usage of Subprograms

The following table refers to the kinds of arguments that can be provided to a Statement Function or a FUNCTION subprogram when it is used in an arithmetic expression, used with a Built-In Function, and used in the CALL statement to a SUBROUTINE subprogram.

Arguments	Statement Function	Built-In Function	FUNCTION Sub-program	SUB-ROUTINE Sub-program
Constant	X	X	X	X
Simple Variable	X	X	X	X
Subscripted Variable	X	X	X	X
Array Name			X	X
Arithmetic Expression	X	X	X	X
External Name			X	X

Defining Statement Functions

A Statement Function is defined by a single arithmetic statement and is valid only in the program in which it appears. It cannot be used by another program or subprogram.

General Form
a = b a represents a function name followed by a pair of parentheses enclosing its arguments. These arguments must be unique, simple variables, separated by commas. b represents an arithmetic expression that does not contain subscripted variables. This expression may also contain other function names that must have been previously defined in the program.

EXAMPLES

```
FIRST(X) = A*X+E
SECOND(X, B) = A*X+B
THIRD(D) = FIRST (E)/D
FOURTH(F, G) = SECOND (F, TIIIRD (G))
FIFTH(I,A) = 3.0*A**I
```

A maximum of 30 variables appearing in "b" may be stated in "a" as arguments. The arguments are dummy names that serve to indicate the type of variable. Those variables included in "b" that are not specified in "a" as arguments are parameters of the function. Thus, in the first example above, A and E are parameters, X is the argument of the function FIRST.

All Statement Function definitions must precede the first executable statement of the source program. There is no limit to the number of Statement Functions.

A typical use of a Statement Function previously defined under "Examples" is:

```
C = R*SECOND(C+D(1, 2),4.0)
```

X of the statement definition takes the value of the arithmetic expression C+D(1,2) and B takes the value 4.0.

Defining Subprograms

The method of defining each class of subprogram is described below.

Built-In Function

Built-In Functions are subprograms that are part of the System Library and are predefined.

General Form
Name (a_1, a_2, . . . , a_n) Name is the name of the function. The names are predefined and are listed in Figure 5. The arguments, a_1, a_2, . . . , a_n, may be arithmetic expressions, subscripted or simple variables, constants, or other Built-In Functions. The number of arguments is specified for each Built-In Function in Figure 5.

A list of all the Built-In Functions supplied is given in Figure 5. Note that the type (real or integer) of each Built-In Function is predefined and cannot be changed by the user. Note also that the type of the arguments is predefined. The core-storage requirements shown in the table are approximations.

To use a Built-In Function, simply use the function name with the appropriate arguments in an arithmetic statement. For example:

```
ROOT1 = ( -B+SQRT (B**2-4.0*A*C))/2.0*A
```

A Built-In Function name may be used as the argument of another Built-In Function. For example, the following is valid:

```
A = ABS (AMAX1 (COS(ALOG(A)),SQRT
    (AMIN1 (C,D,E))))
```

Name	Description	No. of Arguments	Type of Arguments	Type of Function	Core-Storage Requirements
SIN	Trigonometric sine Argument must be less than 10000. radians in absolute value.	1	Real	Real	1155
COS	Trigonometric cosine Argument must be less than 10000. radians in absolute value.	1	Real	Real	
ALOG	Natural logarithm Argument must be greater than zero.	1	Real	Real	1051
EXP	Argument power of e (i.e., e^x) Argument must be less than 225.	1	Real	Real	1194
SQRT	Square root For negative arguments, the square root of the absolute value is calculated and an error message is given on the Standard Print Unit. The user who desires the square root of the absolute value of a number can avoid receiving the error message by writing SQRT (ABS (A)) instead of SQRT(A).	1	Real	Real	739
ATAN	Arc tangent	1	Real	Real	1320
ABS	Absolute value	1	Real	Real	51
IABS		1	Integer	Integer	62
FLOAT	Convert integer argument to real	1	Integer	Real	237
IFIX	Convert real argument to integer	1	Real	Integer	239
AINT	Take the integral part of a real number (sign of argument times largest	1	Real	Real	209
INT	integer $\leqq$ 1 argument 1)	1	Real	Integer	130
AMOD	Argument 1 modulus argument 2	2	Real	Real	348
MOD	The absolute value of the modulus is used if the second argument should be negative. The result (residue) will be non-negative and less than the modulus. If the modulus is zero, a zero answer is obtained and an error message is given on the Standard Print Unit.	2	Integer	Integer	247
AMAX0		$\geqq$2	Integer	Real	204
AMAX1	Maximum value of two or more arguments	$\geqq$2	Real	Real	178
MAX0		$\geqq$2	Integer	Integer	180
MAX1		$\geqq$2	Real	Integer	186
AMIN0		$\geqq$2	Integer	Real	204
AMIN1	Minimum value of two or more arguments	$\geqq$2	Real	Real	186
MIN0		$\geqq$2	Integer	Integer	179
MIN1		$\geqq$2	Real	Integer	122
SIGN	Absolute value of argument 1 times sign of argument 2	2	Real	Real	78
ISIGN		2	Integer	Integer	62
DIM	Argument 1 minus the lesser of argument 1 and argument 2	2	Real	Real	148
IDIM		2	Integer	Integer	139
SLITE	Simulated Sense Light Manipulation and Testing (see "Machine Indicator	1	Integer		14
SLITET	Tests")	2	Integer		237
EXIT	Exit subroutine (see "EXIT Subroutine")	0			7
DVCHK	Division Overflow (see "Machine Indicator Tests")	1	Integer		75
OVERFL	Arithmetic Overflow (see "Machine Indicator Tests")	1	Integer		75

Figure 5. Built-In Functions

FUNCTION Subprogram

General Form
FUNCTION name (a_1, a_2, . . . , a_n) REAL FUNCTION name (a_1, a_2, . . . , a_n) INTEGER FUNCTION name (a_1, a_2, . . . , a_n) name is the symbolic name of the function. The arguments, a_1, a_2 . . . , a_n, must be nonsubscripted variable names, or array names, or the dummy names of SUBROUTINE or other FUNCTION subprograms. There must be at least one argument in a FUNCTION subprogram.

The type of function may be explicitly stated by the inclusion of the word REAL or INTEGER before the word FUNCTION, as shown above.

EXAMPLES

```
FUNCTION ARCSIN(RADIAN)
REAL FUNCTION IROOT (A, B, C)
INTEGER FUNCTION CONST (INT, J)
```

The FUNCTION subprogram is similar to the Statement Function in that it returns only one value to the calling program; it is similar to the SUBROUTINE subprogram in that it may consist of many statements.

No card should precede the FUNCTION statement.

The FUNCTION subprogram may contain any FORTRAN statement except a SUBROUTINE statement or another FUNCTION statement.

The arguments of the FUNCTION subprogram may be considered to be dummy variable names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the main program. The actual arguments must correspond in number, order, and type to the dummy arguments.

The relationship between variable names in the main program and the dummy names in the FUNCTION subprogram is illustrated in the following example:

```
MAIN PROGRAM                FUNCTION SUBPROGRAM
                            FUNCTION SOMEF (C, B)
        .                           .
        .                           .
        .                           .
A = SOMEF (B, C)            SOMEF = B/C
        .                           .
        .                           .
        .                           .
                            RETURN
                            END
```

In the example, the value of variable B of the main program is used in the subprogram as the value of the dummy variable C, and the value of C is used in the subprogram for the value of B. Thus, if the value of B is 10.0 and the value of C is 5.0, the value returned by the subprogram is 0.5 (not 2.0).

When a dummy argument is an array name, an appropriate DIMENSION statement (see "The Specification Statements") must also appear in the FUNCTION subprogram. The corresponding actual argument must be an array name that appears in a DIMENSION or COMMON (With Dimensions) statement in the main program.

None of the dummy names in the subprogram may appear in an EQUIVALENCE or COMMON statement in the FUNCTION subprogram (see "The Specification Statements").

The value of the formal arguments of a FUNCTION subprogram must not be redefined in the subprogram. That is, they must not appear on the left side of an arithmetic statement, nor in an input list, nor as the index in a DO statement. Variables that appear in eommon storage may not be redefined either. For example, the following violates this rule:

```
FUNCTION SAM (A, B, K)
COMMON J
J=J+1
K=J
```

The FUNCTION subprogram must return control to the main program with a RETURN statement.

The name of the function must appear at least once as the variable name on the left side of an arithmetic statement or in an input statement. For example:

```
MAIN PROGRAM                FUNCTION SUBPROGRAM
                            FUNCTION CALC (A,B,J)
        .                           .
        .                           .
        .                           .
                            I=J*2
ANS=ROOT1*CALC(X,Y,I)               .
        .                           .
        .                           .
        .                           .
                            CALC=A**I/B
                                    .
                                    .
                                    .
                            RETURN
                            END
```

In this example, the values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J, respectively. The value of CALC is computed and this value is returned to the main program where the value of ANS is computed.

END AND RETURN STATEMENTS

Note that all of the preceding examples of FUNCTION subprograms contain both an END and at least one RETURN statement. The END statement specifies, for the processor, the end of the subprogram; the RETURN statement signifies a logical conclusion of the computation and returns any value computed and control to the calling program. There may, in fact, be more than one RETURN statement in a FUNCTION subprogram.

For example:

```
FUNCTION DAV (D, E, F)
IF (D.GT.0.1) GO TO 2
        .
        .
        .
IF (E.LT.F) GO TO 3
        .
        .
        .
2 DAV = .....
        .
        .
        .
  RETURN
3 DAV = .....
        .
        .
        .
  RETURN
  END
```

SUBROUTINE Subprogram

General Form
SUBROUTINE name (a_1, a_2, . . . , a_n) name is the name of this subprogram. a_1, a_2, . . . , a_n are the arguments. (There need not be any.) Each argument used must be a nonsubscripted variable name or array name, or the dummy name of another SUBROUTINE or FUNCTION subprogram.

EXAMPLES

```
SUBROUTINE MATMPY (A, N, M, B, L, J)
SUBROUTINE QDRTIC (B, A, C, ROOT1, ROOT2)
```

No card should precede the SUBROUTINE statement.

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. Any arguments so used must appear on the left side of an arithmetic statement or in an input list within the subprogram.

The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. The actual arguments must correspond in number, order, and type to the dummy arguments.

When a dummy argument is an array name, a DIMENSION statement must appear in the SUBROUTINE subprogram. The corresponding actual argument in the CALL statement must also be a dimensioned array name.

None of the dummy arguments may appear in an EQUIVALENCE or COMMON statement in the SUBROUTINE subprogram.

Like the FUNCTION subprogram, the SUBROUTINE subprogram must return control to the calling program by a RETURN statement.

An END statement is also required.

Subprogram Names as Arguments — The EXTERNAL Statement

Subprogram names may be used as the actual arguments in the calling program. In order to distinguish these subprogram names from ordinary variables when they appear in an argument list, their names must appear in an EXTERNAL statement (see "The Specification Statements").

The CALL Statement

The CALL statement is used only to call a SUBROUTINE subprogram.

General Form
CALL name (a_1, a_2, . . . , a_n) name is the symbolic name of a SUBROUTINE subprogram. a_1, a_2, . . . , a_n are the actual arguments that are being supplied to the SUBROUTINE subprogram.

EXAMPLES

```
CALL MATMPY (X, 5, 40, Y, 7, 2)
CALL QDRTIC (X, Y, Z, ROOT1, ROOT2)
```

The CALL statement transfers control to the SUBROUTINE subprogram and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement. The arguments in a CALL statement may be any of the following: any type of constant, any type of subscripted or nonsubscripted variable, an arithmetic expression, the name of a subprogram.

The arguments in a CALL statement must agree in number, order, type and array size with the corresponding arguments in the SUBROUTINE subprogram.

Machine Indicator Tests

The 1410/7010 FORTRAN language provides machine indicator tests even though machine components referenced by the tests do not physically exist. The machine indicators, described below, are simulated by SUBROUTINE subprograms located in the System Library.

To use any of the following machine indicator tests, the user supplies the proper arguments and writes a CALL statement. In the following listing, i is an integer expression, j is an integer variable.

GENERAL FORM	FUNCTION
SLITE (i)	If i=0, all sense lights are turned off. If i=1, 2, 3, or 4, the corresponding sense light is turned on.
SLITET (i, j)	Sense light i (1, 2, 3, or 4) is tested and j is set to "1" or "2" if i is on or off, respectively. After the test, sense light i is turned off.
OVERFL (j)	This indicator is on if an arithmetic operation with real variables and constants results in an overflow condition; that is, if an arithmetic operation (of type real) produced a result whose value is greater than $(1-10^{-f})\times 10^{99}$. If the indicator is on, j is set to "1"; if off, j is set to "2." The indicator is set to off after the test is made.
DVCHK (j)	This indicator is set on if an arithmetic operation with real constants and variables results in the attempt to divide by zero; j is set to "1" or "2" if the indicator is on or off, respectively. The indicator is set to off after the test is made.

EXAMPLES

```
CALL SLITE (3)
CALL SLITET (K*J, L)
CALL OVERFL (J)
CALL DVCHK (I)
```

As an example of how the sense lights can be used in a program, assume that the statements CALL SLITE (I) and CALL SLITET (I, KEN) have been written. Further assume that it is desired to continue with the program if sense light I is on and to write results if sense light I is off. This can be accomplished using the Relational IF statement or a Computed GO TO statement, as follows:

```
      .
      .
IF (KEN. EQ. 2) WRITE (3, 26) (ANS(K), K=1, 10)
      .
     or
      .
      .
GO TO (6, 17) KEN
17 WRITE (3, 26) (ANS(K), K=1, 10)
 6
      .
      .
```

EXIT Subroutine

A CALL to the EXIT subprogram, located in the System Library, terminates the execution of the program and returns control to the Monitor. The EXIT subprogram and the STOP statement produce identical results.

General Form
CALL EXIT

The specification statements provide information concerning storage allocation and the variables used in a program. The specification statements are the DIMENSION statement, the COMMON statement, the EQUIVALENCE statement, and the Type statements.

DIMENSION Statement

General Form
DIMENSION $v_1(i_1), v_2(i_2), \ldots, v_n(i_n)$ $v_1, v_2, \ldots, v_n$ are the names of arrays. $i_1, i_2, \ldots, i_n$ are each composed of 1, 2, or 3 unsigned integer constants, where each integer specifies the maximum value of that subscript.

EXAMPLE

```
DIMENSION A(10), B(5, 15), C(9, 9, 9)
```

The DIMENSION statement provides information to allocate storage for arrays in an object program. It defines the maximum size of each array listed.

Each variable that appears in subscripted form in a source program must appear in a DIMENSION statement contained within the source program. There is one exception to this rule: when the dimension information for the array is given by a COMMON statement. See "COMMON (With Dimensions) Statement" in this section.

The required location of DIMENSION statements appears later in this section under "Order of Specification Statements."

A maximum of 200 names may be dimensioned. Within this limit, (1) a single DIMENSION statement may specify the size of any number of arrays, and (2) a program may have any number of DIMENSION statements.

Dummy variable array names in subprograms also require dimension information in the subprogram.

COMMON Statement

General Form
COMMON a, b, ... a, b, ... are variable or array names.

The COMMON statement refers to a common area of core storage. Variables or arrays that appear in main programs and subprograms can be made to share the same storage locations by use of the COMMON statement. For example, if one program has the statement COMMON A and a second program contains the statement COMMON X, variables or arrays A and X will occupy common storage locations.

The required location of COMMON statements appears later in this section under "Order of Specification Statements."

A maximum of 100 names may be declared in COMMON by means of this statement and the COMMON (With Dimensions) statement.

Within a specific program or subprogram, variables and arrays are assigned storage locations in the sequence in which their names appear in a COMMON statement. Subsequent sequential storage assignments within the same program or subprogram are made with additional COMMON statements.

As an example, if the main program contains the statement

```
COMMON A, B, C
```

and a subprogram contains the statement

```
COMMON X, Y, Z
```

then A, B, and C are assigned sequential locations, as are X, Y, and Z. Furthermore, A and X will occupy the same location, as will B and Y, and also C and Z.

Names declared in COMMON must agree, respectively, in mode. In the preceding example, A and X are real, as are B and Y, and C and Z.

A dummy variable can be used in a COMMON statement to establish shared locations of variables that would otherwise occupy different locations. For example, the variable Y can be assigned to the same location as the variable C of the previous example with the following statement

```
COMMON Q, R, Y
```

where Q and R are dummy names that are not used elsewhere in the program.

Redundant COMMON entries are not allowed. For example, the following is *invalid*:

```
Common A, B, C, A
```

COMMON (With Dimensions) Statement

General Form
COMMON $v_1(i_1), v_2(i_2), \ldots, v_n(i_n)$ $v_1, v_2, \ldots, v_n$ are the names of arrays. $i_1, i_2, \ldots, i_n$ are each composed of 1, 2, or 3 unsigned integer constants, where each integer specifies the maximum value of that subscript.

EXAMPLE

```
COMMON A(10), B(5,15, C(5,5,5)
```

This form of the COMMON statement, besides performing the functions discussed previously for the COMMON statement, performs the additional function of specifying the size of arrays.

The required location of COMMON (With Dimensions) statements appears later in this section under "Order of Specification Statements."

A maximum of 100 names may be declared in COMMON by means of this statement and the COMMON statement.

NOTE: A single COMMON statement may contain variable names, array names, and dimensional array names. For example, the following is valid:

```
DIMENSION B(5,15)
COMMON A,B,C(9,9,9)
```

EQUIVALENCE Statement

General Form
EQUIVALENCE (a,b, . . .), (d,e, . . .), . . . a,b,d,e, . . . are simple variables or subscripted variables. Subscripted variables must have single subscripts only and these subscripts must be integer constants.

EXAMPLE

```
EQUIVALENCE (A(1),B(1),C(5)),(D(17),E(3)),(I,J)
```

The EQUIVALENCE statement controls the allocation of core storage by causing two or more variables to share the same core storage location.

Each pair of parentheses in the list encloses the names of two or more variables to be stored in the same location during execution of an object program. These variables *must* be of the same type and must not be inconsistent in relative core-storage locations. For example, EQUIVALENCE (A(4),C(2),D(1)),(A(2),D(2)) is invalid.

The required location of EQUIVALENCE statements appears later in this section under "Order of Specification Statements."

Any number of list items may be given in a single EQUIVALENCE statement.

In the first example, the A, B, and C arrays are to be allocated to core storage so that the elements A(1), B(1), and C(5) are to occupy one location. In addition, D(17) and E(3) are to share another location, as are I and J.

In the second example if A(4), C(2), and D(1) are made equivalent, an equivalence is set up among elements of each row below.

```
A(1)
A(2)
A(3)      C(1)
A(4)      C(2)      D(1)
A(5)      C(3)      D(2)
  .         .         .
  .         .         .
```

Thus, D(2) must not be made equivalent to A(2). EQUIVALENCE (A(3),A(4)) also is invalid.

Variables or arrays that are not mentioned in an EQUIVALENCE statement are assigned unique locations. The sharing of storage locations requires a knowledge of which FORTRAN statements cause a new value to be stored at a location.

Execution of an Arithmetic Statement stores a new value at the location specified by the variable name at the left of the equal sign.

Execution of a DO Statement changes the index each time the program passes through the repetition of the DO.

Execution of a READ Statement stores new values at the locations specified by the variable names in the list.

Execution of a CALL Statement stores the values of the arguments supplied by the calling program and may also affect variables in COMMON.

The user can make double-subscripted variables equivalent by noting the following:

In the EQUIVALENCE statement, he must refer to double-subscripted variables as if they were single-subscripted variables.

EXAMPLE

A(2,2) and B(2,2) would be defined in core storage as:

```
A(2,2) A(1,2) A(2,1) A(1,1)
B(2,2) B(1,2) B(2,1) B(1,1)
```

Since there are four elements in each array, the user can refer to each array as follows when setting up the EQUIVALENCE statement:

```
A(4), A(3), A(2), A(1)
B(4), B(3), B(2), B(1)
```

For example, A(1,2) and B(2,1) can be made equivalent by the following:

```
EQUIVALENCE (A(3), B(2))
```

NOTE: When referring to a subscripted variable in the EQUIVALENCE statement, the user must refer to the variable as it was dimensioned (double-subscripted) — otherwise the statement will be flagged as an error.

COMMON and EQUIVALENCE Statements — Special Considerations

No two elements that appear in a COMMON statement may be made equivalent. Both of the following examples are *invalid*:

```
COMMON A,B                COMMON A,B
EQUIVALENCE (A,B)         EQUIVALENCE (A,R);
                            (R,D),(D,B)
```

EQUIVALENCE statements may extend the size of the COMMON area. For example, the following is *valid*:

```
DIMENSION C(4)
COMMON A,B
EQUIVALENCE (B,C(2))
```

It would produce the following relationship in the COMMON area:

```
A       C(1)
B       C(2)
        C(3)
        C(4)
```

The following is an example of an *invalid* set of statements:

```
DIMENSION C(4)
COMMON A,B
EQUIVALENCE (A,C(2))
```

It would imply the following relationships in the COMMON area:

```
        C(1)
A       C(2)
B       C(3)
        C(4)
```

Thus as shown above, the COMMON statement determines the first element that is to appear in the COMMON area; the EQUIVALENCE statement may not change the position of this element.

Type Statements (INTEGER, REAL, EXTERNAL)

General Form
INTEGER a,b,c, . . . REAL a,b,c, . . . a,b,c, . . . are variable, Statement Function, or FUNCTION subprogram names appearing in a program or subprogram. EXTERNAL x,y,z, . . . x,y,z, . . . are subprogram names used as arguments of other subprograms called by the program.

EXAMPLES

```
INTEGER DAV, ZZZ, LYSL, JOB
REAL IAM, LEG, KKKK
EXTERNAL SIN, MATMPY
```

The REAL and INTEGER statements explicitly define the type (real or integer) of variable, Statement Function, or FUNCTION subprogram. In the first example, the variable DAV implicitly would be a real variable, but the explicit statement causes it to be handled as an integer variable in the program. The appearance of a name in either of these statements overrides any implicit-type specification.

A program using the names of other FUNCTION or SUBROUTINE subprograms as arguments requires an EXTERNAL statement. The statement distinguishes the names of subprograms external to the calling program from the variables of the calling program. For example, assume both SOMEF and OTHER are subprograms. If A = SOMEF (OTHER, B, C) + B appears in a program, the Type statement EXTERNAL OTHER is required in the program.

Similarly, if CALL SOMEF (B, C, OTHER) appears in a program, the Type statement EXTERNAL OTHER is required.

Type statements must precede any other specification statements and all executable statements in the source program.

A name may appear in two Type statements only if one of the statements is EXTERNAL.

Order of Specification Statements

All Specification statements must precede the first executable statement of the source program. The Specification statements must also precede all Statement Function definition statements, and *must* appear in the following order:

```
Type Statements (REAL, INTEGER, EXTERNAL)
DIMENSION
COMMON
EQUIVALENCE
(Follow with FORMAT statements, then Statement Functions.)
```

PART 2 — FORTRAN AS AN OPERATING SYSTEM COMPONENT

Monitor Card to Execute FORTRAN

The EXEQ card is a Monitor control card that causes a program to be loaded and executed. When the first operand of this card is FORTRAN, the FORTRAN processor is loaded and a source program is compiled. Other Monitor control cards required to process a job are explained in the publication *IBM 1410/7010 Operating System; System Monitor*, Form C28-0319.

The EXEQ card format and the operands available to the user desiring to compile a FORTRAN source program are explained below:

EXEQ CARD FORMAT EXAMPLES

```
6       16     21
MON$$   EXEQ   FORTRAN,SOF,SIU,7,12,PCH,FLT,NAMEX
MON$$   EXEQ   FORTRAN,MJB,SIU,7,,PCH,,MAINPROGRM
MON$$   EXEQ   FORTRAN,,,13
```

Columns 6-10 contain the characters MON$$ to identify the card as one directed to the System Monitor.

Columns 16-20 contain the letters EXEQ.

Columns 21-72 may contain up to eight operands.

These standard rules for operands apply: operands must be separated by a comma; operands cannot contain blanks; an intentionally omitted operand must be indicated by placing a comma adjacent to the preceding comma (except when the omitted operand is the last operand used).

The first three operands are required by the System Monitor and must either be included or their omission indicated by a comma. The fourth and following operands are read by the FORTRAN processor. An invalid parameter or error following the third parameter causes (1) a diagnostic message number to be printed immediately following the title line of the source program diagnostic listing, and (2) assumed operands indicated below to be used in the compilation in place of the erroneous operand and all subsequent operands on the card. Compilation proceeds.

The eight operands must appear in the following listed order:

OPERAND NO.	OPERAND AND MEANING
1	FORTRAN must be specified.
2	SOF or MJB — This operand specifies the file containing the FORTRAN compiler. SOF is the System Operating File; MJB is the Job file. If this operand is omitted, the compiler is assumed to be on the SOF.
3	SIU, AIU, or any work (MWn) or reserve (MRn) tape unit. If this operand is omitted, the source statements must immediately follow this EXEQ card on the unit containing this card—either the Standard Input Unit (SIU), or the Alternate Input Unit (AIU).
4	Real number precision (f) is specified by a one- or two-digit number from 3 through 18. Numbers 3 and 03 are acceptable, but 003 is not acceptable. If the operand is omitted, the FORTRAN processor assumes 8.
5	Integer precision (k) is specified by a one- or two-digit number from 3 through 20. If the operand is omitted, the processor assumes 5.
6	PCH — If this operand is used, the object program is put on the Standard Punch Unit or is written on the magnetic tape unit assigned to perform functions of the Standard Punch Unit. If the compiled program is not to be executed immediately (that is, not a compile-and-go operation), PCH must be specified to produce an object deck.
7	FLT — This operand must be specified if the object program is to be run on an IBM 7010 equipped with the Floating-Point Arithmetic feature. The operand must be omitted (except for the trailing comma) for all other IBM 7010 Systems and all IBM 1410 Systems.
8	Program name. This operand, which can be one to ten alphameric characters with the first character alphabetic, defines the title for the main program to be processed. If this operand is omitted, the processor assigns the title MAINPGM. When subprograms are compiled, this operand is ignored.

Control Card Requirements

The sequence of the appropriate Monitor and Linkage Loader control cards needed to compile and execute the FORTRAN program is shown in Figure 6.

Figure 7 shows the sequence of the appropriate Monitor and Linkage Loader control cards necessary to execute the FORTRAN object program.

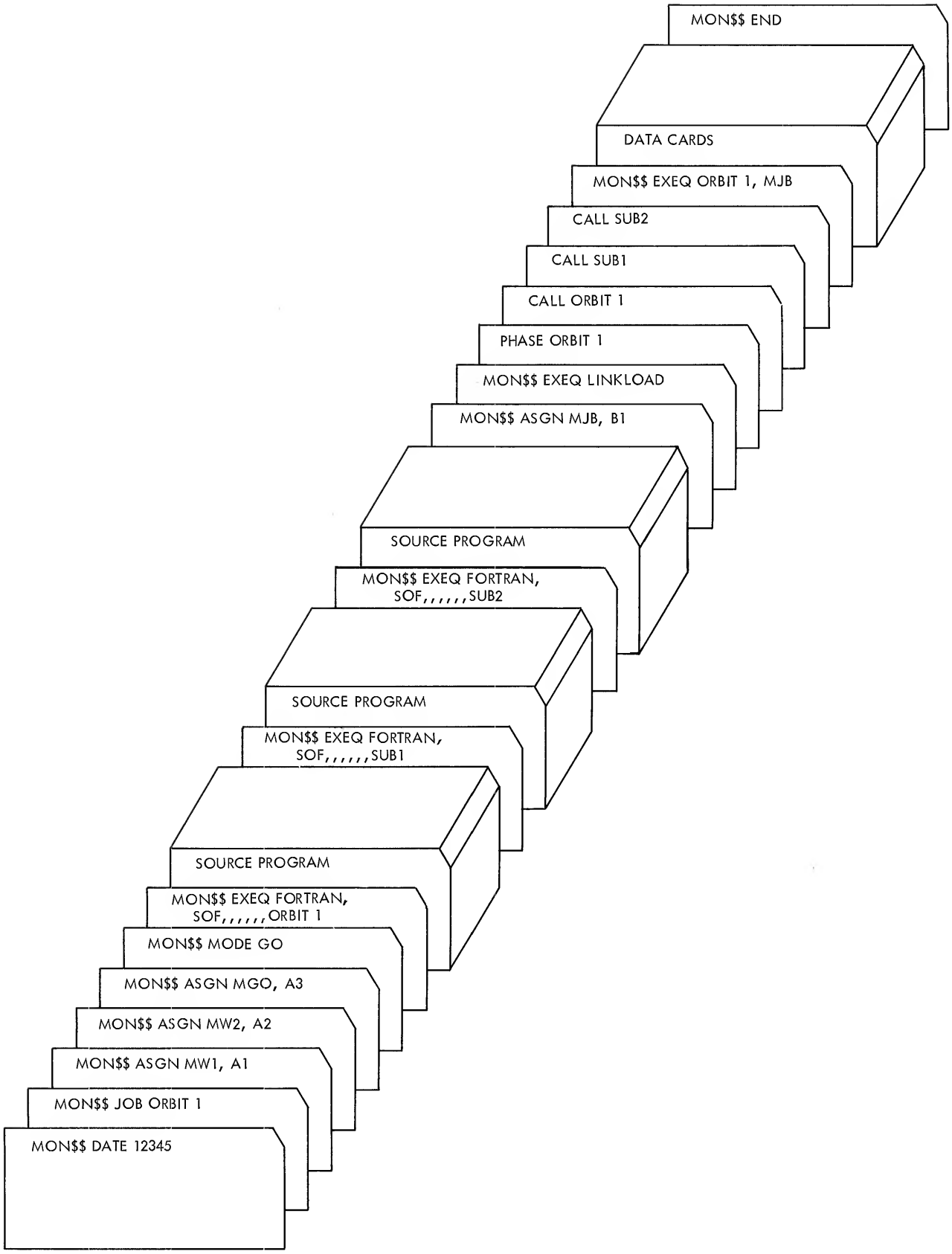


Figure 6. Sample Control Cards for a Compile-And-Go Operation

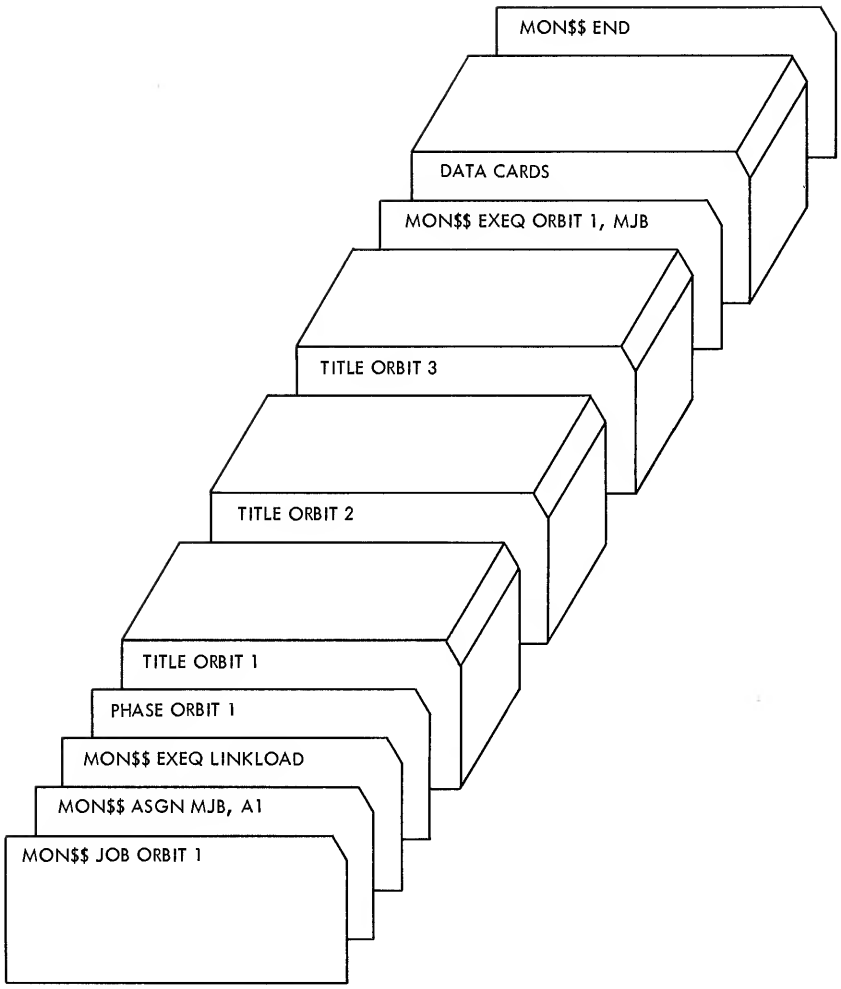


Figure 7. Sample Control Cards for Execution of a FORTRAN Object Program

TITLE Card

The TITLE card (see "TITLE Card" in *IBM 1410/7010 Operating System; System Monitor,* Form C28-0319) defines the name of an object program and is the first record of that program. Produced by the FORTRAN processor during compilation, the TITLE card contains the current date, the program name, and the base (location in core storage) to which the program was compiled. Although such information is included in the TITLE card produced by the Autocoder processor, the TITLE card produced by the FORTRAN processor does *not* contain the size of COMMON. Therefore, when processing a FORTRAN program, the Linkage Loader *cannot* consider the size of COMMON in checking to ensure that core storage is not exceeded.

The source program listing produced by a FORTRAN compilation is written on the Standard Print Unit. The two major parts of the listing are (1) the listing of the source program statements with diagnostic messages, and (2) a memory map of the object program.

Source Program Diagnostic Listing

The following items appear with the listing of source program statements:

1. Title Line. Each page of the source program diagnostic listing carries a title line containing the page number.

2. Source Program Statements. All statements are listed without blanks, except FORMAT statements.

3. Message. Diagnostic message numbers are printed to the left of the statements to which the messages apply. The corresponding messages are listed in Part 3 of this manual. The message number is sometimes followed by a number indicating the character position in the printed statement where the error was found. The first character of the statement is considered character number 6. *Each print position should be counted.* Some messages also include the name of the variable or the missing statement number that caused the error.

Diagnostic messages for context errors, such as terminating a DO on a nonexecutable statement, follow the listing of source statements. These messages may include the statement numbers of the affected statements.

A warning message is printed as a three-digit number. Any additional information associated with the message is printed on the following line.

Memory Map

The memory map of the object program is arranged in four pairs of "FORTRAN NAME," "LOCATION" headings across the page as shown in Figure 8. The map includes the addresses, *before relocation,* of constants, variables, and the beginning of coding of executable numbered statements. The items appear in the list in the sequence given below. (Items illustrated in Figure 8 are numbered to correspond with this list.)

1. Names Declared in COMMON. The name is printed to the left of its address.

2. Integer and Real Constants. The value of the constant is printed under "FORTRAN NAME." The constants are shown as they appear in core storage.

3. Integer Variables. Both simple integer variables and integer arrays are listed next. The address printed for an array is its base address. This is the highest core-storage address in the array and corresponds to the address of the first element. Since some of these variables may have been brought into COMMON by equivalence, COMMON addresses may be included.

4. Real Variables. These are mapped in the same manner as integer variables.

5. Statement Numbers. Each statement number is printed with an address. This is the address of the first machine instruction generated by the statement.

Any forward reference to a statement number causes additional printing. A forward reference is a reference (GO TO n, IF (a) n_1, n_2, n_3, etc.) to a statement number that has not previously appeared on a statement.

There are two types of forward references: unconditional and conditional. Unconditional forward references are those for which unconditional branch instructions are generated; e.g., such an instruction is generated for the reference to statement number n in the statement GO TO n. Conditional forward references are those for which some conditional branch instructions are generated; such an instruction (in this instance, Branch if Zero Balance) is generated for the reference to statement number n_2 in the statement IF (a) n_1, n_2, n_3.

At the first unconditional forward reference to a statement number, a memory map entry is printed. This entry contains the statement number to which reference is made, the address of the reference, and the tag FOR. to denote an unconditional forward reference. No memory map entries are printed for subsequent unconditional forward references to the same statement number.

At the first conditional forward reference to a statement number, a memory map entry is printed. This entry contains the statement number to which reference is made, the address of the reference, and the tag CFOR. to denote a conditional forward reference. No memory map entries are printed for subsequent conditional forward references to the same statement number.

Any forward reference to a statement number causes still another memory map entry. This entry is printed

when the statement number appears on a statement; the entry contains the address of the last forward reference (unconditional or conditional), the address assigned to the statement number, and the tag DEF. to denote statement number definition.

There are three possible combinations of forward references and definitions:

1. CFOR., then DEF.
2. FOR., then DEF.
3. CFOR., then FOR., then DEF.

For the CFOR./DEF. and FOR./DEF. combinations, the CFOR. and FOR. entries point directly to the DEF. entry. For the CFOR./FOR./DEF. combinations, the CFOR. entry points to the FOR. entry, and the FOR. entry points to the DEF. entry.

For example (see Figure 8), suppose that at location 01370 there is a forward reference to statement number 301. If the coding for statement 301 begins at location 01997, the following two entries will be made in the printed output:

00301	FOR.	01370
.	.	.
.	.	.
01370	DEF.	01997

6. Program Size and Entry Point. The last line of the printed output states the size of the compiled program, excluding the common data area, and the location of its entry point.

7. Size of COMMON. The size of the COMMON area can be determined by reference to the listing that appeared as output on the SPR as a result of program compilation. The number of characters used is specified at the end of the program, before the memory map.

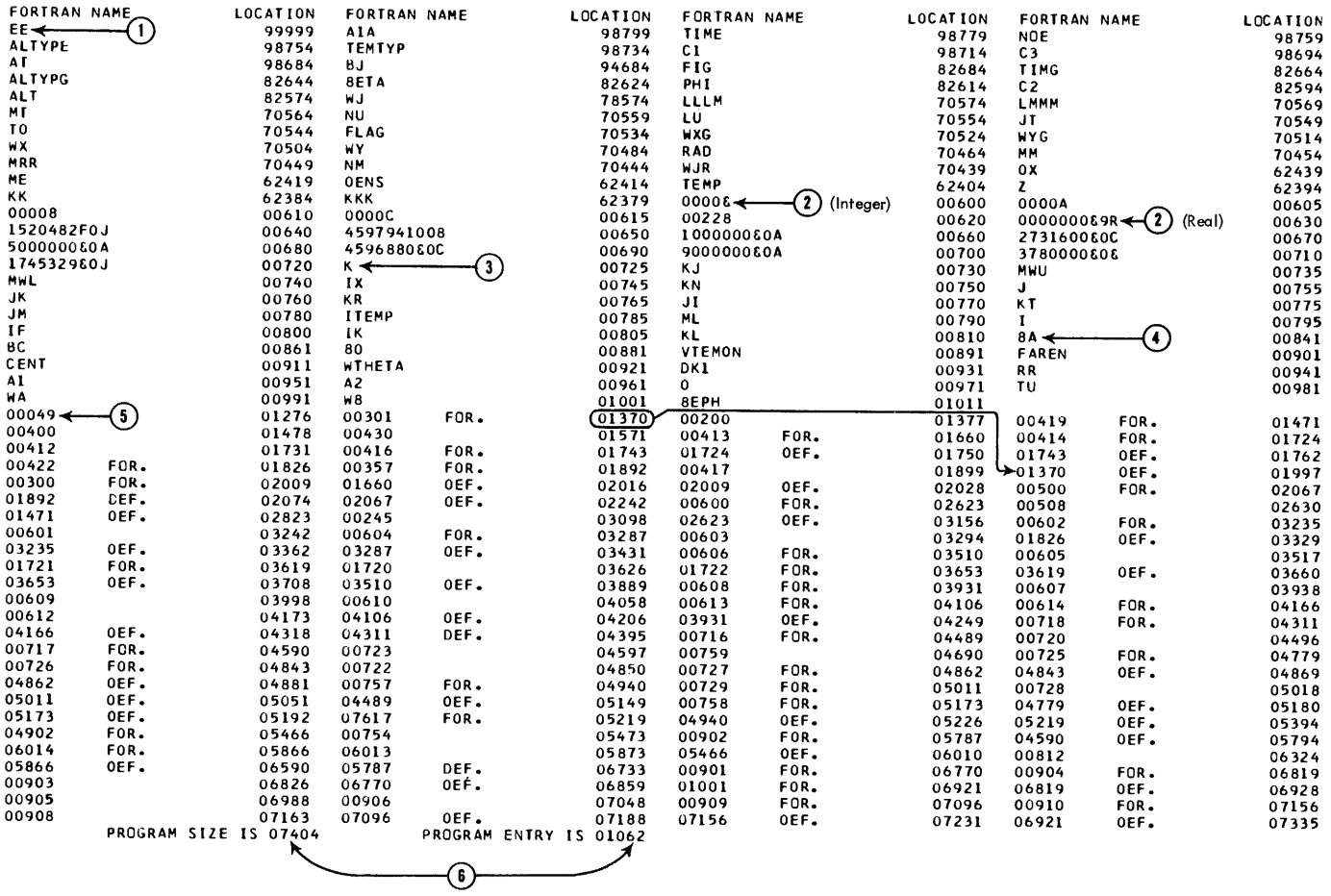


Figure 8. Memory Map of an Object Program

The FORTRAN processor reduces the size and execution time of the object program by avoiding redundant calculations to obtain the memory locations of subscripted variables. Immediate repetition of the same subscript or an equivalent subscript for the same or different arrays is subject to subscript optimization.

A FORTRAN object program reserves up to 100 index cells to hold subscript information. If this limit is exceeded, a diagnostic message is produced. The user may determine that this limit will be exceeded by knowing:

1. When an index cell is reserved for a newly defined subscript expression appearing in a source program statement;
2. Which subscript expressions in the source statements are equivalent and, consequently, use the same index cell; and
3. When a cell is made available during program compilation due to deletion of the subscript expression held in the cell from the list of active subscripts.

Terms Used

An array name in a source statement is subscripted with either a *literal subscript*, which has no variables in it, or a subscript containing *subscripting variables.* For example, A(3,2) has a literal subscript while B(4,J+3,2*M) contains J and M as subscripting variables.

A subscripting variable may have a multiplicative *coefficient* (2 in the preceding example), or an additive *offset* (3 in the example), or both as in C(2*M−1).

A *subscript expression* is the set of three (or fewer) subscripting variables together with four numbers, designated D1 through D4, calculated from the six (or fewer) coefficients and offsets and the three (or fewer) array dimensions declared in the DIMENSION statement. The form of the subscript expression is shown under "Equivalence of Subscript Expressions" later in this section.

Arrays are located in core storage in sequence A(L,M,N) . . . . . A(I,J,K) . . . . . A(3,1,1), A(2,1,1), A(1,1,1) where the last element A(L,M,N) has the low address in core storage and the first element has a higher address in core storage. The *value* of the subscript expression I,J,K is the number of core-storage positions separating element A(I,J,K) from the first element of the array, A(1,1,1).

The *value* of the subscript expression is calculated in two steps.

1. The values of D1 through D4 are calculated by the processor from values known at compilation time.
2. The values of the subscripting variables I,J,K are not known until the program is being executed. During execution, D1 through D4 and the now-known values of the subscripting variables are used to calculate the value of the subscript expression.

The *index cell* contains the value of the subscript expression.

Reserving Index Cells

An index cell is reserved whenever a nonliteral subscript appears in a source statement and is not equivalent to a subscript expression already considered active. The first nine index cells are index registers; up to 91 subsequent cells are pseudo index registers that are defined by the compiler. The points in the source program at which index cells will be reserved by the compiler can be determined by inspecting each statement in order in the source deck.

The appearance of a nonliteral subscript in a source program, such as A(I,J), produces object coding to calculate the value of the subscript expression corresponding to (I,J), and place the value into an index cell. The instruction referring to A(I,J) effectively has an address field with the address of the base of the array A indexed with the proper zones to indicate the index register that is the index cell assigned to (I,J).

An immediate repetition of A(I,J) in the source program produces no coding to set an index cell to the value of the (I,J) subscript expression, since a cell has already been reserved for that value. Conditions under which subscript expressions are equivalent and can use a single index cell are described under the next heading. In general, the equivalence does not require the same array A, but the arrays will have the same number and sizes of dimensions.

An index cell is reserved for each subscript expression that is not equivalent to a subscript expression considered active. A subscript expression is considered active, and an index cell is reserved for its value, until the value of one of the subscripting variables is changed by the program.

Ways in which a subscripting variable may be changed by a program are explained under "Deleting

Subscript Expressions." An example is the arithmetic statement I = I + 1, which alters I.

A literal subscript, as BB(3,65,2), does not require an index cell. The proper address in the array BB is calculated at compilation time for each instruction requiring it.

Equivalence of Subscript Expressions

Rules to determine whether two subscript expressions are equivalent are given in the following paragraphs. Expressions that are not equivalent require different index cells.

The form of the subscript expression is D1, I, D2, J, D3, K, D4, where I, J, and K are the subscripting variables and the D factors are calculated as explained below. Two subscript expressions are equivalent when the subscripting variables appear in the same order and the four D factors of one subscript expression equal, respectively, the four D factors of the other subscript expression.

Consider any array, A, specified as DIMENSION A(x,y,z), where the lower-case x, y, and z are integer, unsigned constants.

A for this explanation may represent either an array of integer elements or real elements,
x is the first dimension,
y is the second dimension,
z is the third dimension.

An element of the array is of the form

$A(c_1*I+c_1',c_J*J+c_J',c_K*K+c_K')$, where
c coefficients are integer constants,
c' terms are positive or negative integer offsets,
I, J, K are subscripting variables.

For the calculation of the D factors, let

w = k if A is an array of integer elements, or
w = f+2 if A is an array of real elements.

The D factors are calculated as follows:

$D1 = c_1*w$
$D2 = c_J*w*x$
$D3 = c_K*w*x*y$
$D4 = (c_1'-1)*w+(c_J'-1)*w*x+(c_K'-1)*w*x*y$

The equation for D1 shows that subscript expressions of the same mode are not equivalent if the c_1 values are not identical.

Note that a subscript expression associated with an array of real elements can be equivalent to a subscript expression associated with an array of integer elements if D1 through D4 of the two expressions are equal.

D4 defines the necessary conditions on the offsets. It shows that two subscript expressions will be equivalent if values of D1 through D3 are equal and the corresponding offsets are identical.

Deleting Subscript Expressions

The following paragraphs give the rules to determine when the value of a subscript expression changes. The change establishes a point of definition at which the index cell associated with that subscript expression is freed. An expression changes when one of the subscripting variables in the expression changes. There are five points of definition of subscripting variables that occur during compilation.

1. A Referenced Statement Number: A statement number to which *control transfers* is a point at which all index cells are freed. Assignment of index cells starts over as subscripting variables are encountered by the processor.

2. A DO Statement: A DO statement, like a referenced statement number, is a point where all index cells are freed. (They are freed since any subscripting variable may change in the range of the DO.)

3. An Arithmetic Statement: If a subscripting variable is on the left side of the equal sign in the arithmetic statement, the only index cells to be freed are those with subscript expressions involving the variable. Thus, the statement I = I + 1 frees the cells of A(I, J) and B(K,I + 3) but not the cell for C(K + 3, J − 6).

4. Input Lists: The appearance of a subscripting variable in the list of a READ statement frees only the index cells associated with that variable.

An implied DO loop in an input list, as in the case of the DO statement, is a point where all index cells are freed.

5. CALL Statements: A SUBROUTINE subprogram may alter its arguments. Consequently, a subscripting variable as an argument of a CALL frees any associated index cells. For example, the statement CALL SUBR (I, J + K) frees any index cells associated with I but does not affect index cells associated with the individual subscripting variable J and K. Release of the index cell is effective after the call so that subscripts appearing in the CALL arguments remain optimized.

A subroutine subprogram may alter COMMON variables. Consequently, the index cells for all subscripting variables in common core storage are freed after the CALL statement.

Since function subprograms do not alter their arguments or variables in common core storage, the appearance of a function call in an arithmetic statement does not free any index cells.

Exceeding the 100 permitted subscript expressions between points of definition (an infrequent occurrence) is called to the user's attention by a diagnostic message printed out during program compilation (see Part 3). Furthermore, a point of definition can be forced at any desired point in the source program.

The statement I = I frees the index cells holding the values of subscript expressions involving I. (See rule 3 above.)

A method that frees all index cells utilize rule 1 above. A statement number to which control never transfers can be placed in a Computed GO TO statement, as GO TO (7,10),K where K is always 2. Statement 7 now becomes a point of definition at which all index cells are freed.

Dictionary Space Requirements

The dictionary is a work area within the FORTRAN processor. The maximum dictionary space available to the processor depends on the machine core-storage size. The dictionary occupies all space from the end of the FORTRAN processor to the memory size specified at /AMS/ in the Resident Monitor. Refer to the publication *1410/7010 Operating System; System Monitor* for information concerning /AMS/.

The space required by each variable or number in a source program is as follows:

TYPE OF VARIABLE OR NUMBER	POSITIONS OF CORE STORAGE REQUIRED
Dimensioned variable	40
Equivalenced, non-dimensioned variable	20
Non-equivalenced, simple variable	10
Real constant	10+f+2, where f is the real precision.
Integer constant	10+k, where k is the integer precision.
Statement number	10

The flexibility of the Autocoder language can be incorporated in a FORTRAN program by means of Autocoder subprograms. Such a subprogram may be desired under these circumstances.

1. Where the number of arguments (and, therefore, the length of the calling sequence) may vary, as for a general-purpose subprogram;

2. Where an input/output manipulation or communication with the Resident Monitor is not available to the FORTRAN programmer;

3. Where charaeter manipulation is necessary,

4. Where data requires specialized decoding or rearrangement before it can be used with FORTRAN statements.

Both FUNCTION subprograms and SUBROUTINE subprograms may be written in Autocoder. The subprogram is assembled by the 1410/7010 Autocoder processor and incorporated into the System Library, where it is available to be combined with FORTRAN programs.

Understanding of the publications, *IBM 1410* or *IBM 7010 Principles of Operation* (Form A22-0526 or A22-6726, respectively) and *IBM 1410/7010 Operating System; Autocoder*, Form C28-0326, is necessary to write Autocoder subprograms.

Calling Sequences

Subprogram arguments can be madc available either by:

1. Declaring the arguments in the common data area both in the calling programs and in the called subprogram, or

2. Listing the arguments following the name of the function in an arithmetic statement, as A=SOMEF(B,C), or following the name of the subprogram in the CALL statement, as CALL other(D,E). A FUNCTION subprogram requires at least one argument following the name of the function in an arithmetic statement.

The call to a FUNCTION or SUBROUTINE subprogram generates a calling sequencc when the program is compiled. The calling sequence begins with a branch to the FUNCTION or SUBROUTINE subprogram followed by a series of address constants (one per argument), and concludes with a NOP. Manipulation of the arguments within an Autocoder subprogram is performed by moving the address constants into the operands of succeeding instructions which can:

1. "Work on" the argument while leaving it in the calling program, or

2. Bring the argument into a work area defined in the subprogram.

The following table summarizes the form and significance of the generated address constants for various types of arguments.

TYPE OF ARGUMENT, REAL OR INTEGER	FORM AND SIGNIFICANCE OF ADDRESS CONSTANT
Simple variable	Address of the variable
Constant	Addrcss of the eonstant
Subscripted variable	Address of the subscripted variable, possibly indexed
Array	Address of the base of the array; i.e., address of ARRAY (1, 1, 1)
Expression	Address of the value of the eomputed expression indexed by X1
Name of a subprogram	Address of the subprogram entry point

EXAMPLE

Assume X is an array and Y is a simple real variable. Also assume the FORTRAN processor assigns index register 4 to the subscript expression of array X. CALL SAMPLE (X(I,J),Y) typically compiles into thc following calling scquence in the calling subprogram.

EFFECTIVE AUTOCODER STATEMENTS

OPERATION CODE	OPERAND
DCWS	SAMPLE
DCW	X+X4
DCW	Y
NOP	

Index Register Requirements

Requirements and conventions for use of the index registers are summarized in the following table.

INDEX REGISTER	AVAILABILITY	USE AND LIMITATIONS
X1	Available if eontents saved and restored	X1 must be preserved for the proeessor and may be tagged on any argument address. After bringing in arguments, X1 may be saved, used, and restored like X4 through X12.
X2, X3	Available	Contents will be destroyed by ealls to most library routines.
X4 through X12	Available if eontents saved and restored	These registers contain subscript index values and may have been speeified on any

INDEX REGISTER	AVAILABILITY	USE AND LIMITATIONS
X13	Available	of the arguments to the subprogram. Contents must be saved in a temporary location and restored before exiting from the subprogram. Contents will be destroyed by calls to library routines or Monitor functions. X13 must not be left with a negative sign when exiting from the subprogram.
X14, X15	Not available	

Writing the Subprogram

Basic Requirements

Each subprogram must be preceded by an Autocoder TITLE card. The operand of this statement is the title of the function or subroutine subprogram. FORTRAN rules for assigning names to integer and real variables must be followed in naming the subprograms (see Part 1 of this manual).

The first instruction should store the B-address register contents. Index register 13 is conventionally used for this purpose. The entry point of the subprogram must be its first position.

The move instructions that place the address constants into the operands of subsequent instructions, including return to the desired position in the calling program, usually are placed next. This is shown in "Examples of Autocoder Subprograms" appearing later in this section.

If any of index registers 4 through 12 are to be used in the subprogram, the contents must be saved after the address constants are moved and before the index register is used by the subprogram. Contents must be restored to the index register before returning to the calling program.

Word lengths of integers and real numbers must correspond to those of the using programs.

A label that is a linkage symbol of the form IBXX/ (where "IB" are the first two characters) is not permitted in the subprogram.

A word mark must follow the last executable instruction of the subprogram.

Handling Real Arguments

If the arguments are real numbers, a move instruction (MLCWA) can bring in the exponent and a chained move (again, MLCWA) can bring in the fraction. This is shown in Example 1 of "Examples of Autocoder Subprograms." (The address constant of the calling sequence gives the units position of the exponent.)

If the address constants are moved into the operands of floating-point instructions, both the exponent and fraction are automatically handled by the floating-point instructions. A DCWS to the floating-point interpretive subroutine IBINTRP must precede a sequence of floating-point instructions unless the object program is to be run on an IBM 7010 equipped with the Floating-Point Arithmetic feature.

The handling of real variables is illustrated in "Examples of Autocoder Subprograms."

Common Data Area

If arguments are obtained from common data area, the variable must be declared in common core storage within the subprogram by the use of the Autocoder DAV statement.

Using Other Functions

System Library subroutines may be freely used by the subprogram. However, conventional uses of the index registers discussed earlier in this section must be observed. In particular, the floating-point interpretive subroutine (IBINTRP) may be used at will. This subroutine accepts indexed addresses in the floating-point instructions that it interprets.

Any argument in the Autocoder subprogram can be made available for transmission to another subroutine by the placing of its address in a calling sequence to that subroutine. This is shown in Example 2 of "Examples of Autocoder Subprograms."

The value of f or k can be transmitted from the FORTRAN main program to an Autocoder subprogram by the use of system symbols /FLO/ or /FIX/ as operands. This permits generalizing a subprogram that makes use of the values of f and k so that it will accept various values of f and k without subprogram alteration.

Although designed for FORTRAN, Built-In Functions may be adapted for use in Autocoder. The programmer must write the calling sequence. He must use the appropriate parameters (see "Constants"), and define the /FIX/ and /FLO/ symbols. The following cards — to be placed into the relocatable deck of the Autocoder program that is to use the Built-In Function — will define these symbols:

```
1 2          7 8          13 14 16 17 19          72
W            W            W     W     W
 S aaaaa      S 00004      S  ff S  kk S           1
6            16           21                      72
/FLO/        DEFIN        aaaaa+1                 4
6            16           21                      72
/FIX/        DEFIN        aaaaa+3                 4
```

In the card formats shown, the following holds true:

W over S is a word separator.
aaaaa is the load address for the Load record.
ff is the unsigned real word-size constant (it may range from 03 through 18, and includes the exponent).

kk is the unsigned integer word-size constant (it may range from 03 through 20).
/FLO/ and /FIX/ are the system symbols to be defined.
aaaaa+1 is the address assigned to the real word-size constant.
aaaaa+3 is the address assigned to the integer word-size constant.
1 and 4 (in columns 72) are record-type indicators.

Several of the subroutine functions, in turn, call in other subroutines. In such cases, the core-storage requirement is the total of all of the subroutines used. When the Linkage Loader processes the object program, it takes the necessary subroutines from the library and relocates both the object program and the subroutines into an absolute, object-time program. Following is a list of subroutine functions that call in other subroutines:

COS	requires	SIN
SLITE	requires	SLITET
INT	calls	IFIX
MIN1	calls	IFIX
MAX1	calls	IFIX
AMIN0	calls	FLOAT
AMAX0	calls	FLOAT
AMOD	calls	AINT

NOTE: IBINTRP is called by most of the floating-point routines.

There is no limit to the depth to which calls to other subroutines can be nested.

The Autocoder subprogram must not call a subroutine that calls the Autocoder subprogram, or causes the Autocoder subprogram to be called, unless specific arrangements have been made for recursive operation.

Returning Values to Calling Program

Either FUNCTION subprograms or SUBROUTINE subprograms may communicate with the calling program by placing information on work tapes.

FUNCTION SUBPROGRAMS

Core-storage locations 280-299 are reserved for the 20-position accumulator used by the floating-point instructions. The single result of a FUNCTION subprogram, whether that result is type integer or type real, must be right-justified in the accumulator. The name of the function, whether type integer or type real, in the calling program must correspond to the type of result.

FUNCTION subprograms must alter neither the variables that appear in the common data area nor the arguments of the FUNCTION subprogram.

SUBROUTINE SUBPROGRAMS

SUBROUTINE subprograms are permitted to alter variables that appear in the common data area.

SUBROUTINE subprograms also are permitted to alter their arguments. The argument address may be picked from the calling sequence and placed into the B-address of the move instruction or the A-address of the floating-point store (FST) instruction that is to return the value to the calling program.

Examples of Autocoder Subprograms

Example 1

This example illustrates:

1. Moving the real arguments into work areas for use in a subprogram, and
2. Saving the contents of an index register.

Assume that the two real arguments of the example are to be placed in work areas TEMP1 and TEMP2 for use in the subprogram. The arguments are eight decimal digits plus the two-digit exponent in length. Assume that X4 is assigned by the FORTRAN processor when compiling the calling sequence and that X4 is also required by the subprogram. Also assume that X13 will not be altered within the subprogram.

CALLING PROGRAM	EFFECTIVE AUTOCODER CALLING SEQUENCE OPERATION CODE	OPERAND
.		
.		
.		
CALL SAMPLE (A(I), B)	DCWS	SAMPLE
.	DCW	A+X4
.	DCW	B
.	NOP	

AUTOCODER SUBPROGRAM

LABEL	OPERATION CODE	OPERAND	COMMENTS
	TITLE	SAMPLE	
	SBR	X13	
	MLCA	4+X13,L1+5	Move address of A into next instruction
L1	MLCWA	0,TEMP1	Move exponent and
	MLCWA		fraction of argument A in chain move
	MLCA	9+X13,L2+5	
L2	MLCWA	0 TEMP2	Move in second argument (B)
	MLCWA		
	MLCA	X4,SAVE4=5	Save X4
	(X4 is now free to use and the arguments have been brought into the subprogram.)		
	.		
	.		
	.		
	Calculations		
	.		
	.		
	.		
	MLCA	SAVE4,X4	Restore X4
	B	11+X13	Return to calling program
	DCW	=8	
TEMP1	DCW	=2	
	DCW	=8	
TEMP2	DCW	=2	
	END		

Example 2

This example illustrates:

1. Calling a function whose name is an argument to this Autocoder subprogram and whose value is a function of the arguments of this Autocoder subprogram; and

2. Return of the externally-computed value, via this Autocoder subprogram, to the calling program.

Assume that the arguments are subscripted variable A(I), expression B + C, external subroutine EXTSUB, and simple variables. The single argument of EXTSUB is the sum of the arguments A(I), B + C, and S. The result of EXTSUB is to be returned to the FORTRAN-language program by altering argument S. Further assume that GT + X1 is the address of the computed expression B + C.

CALLING PROGRAM	EFFECTIVE AUTOCODER CALLING SEQUENCE	
	OPERATION CODE	OPERAND
.	DCWS	SAM
.	DCW	A+X6
EXTERNAL EXTSUB	DCW	GT+X1
.	DCWF	EXTSUB
.	DCW	S
.	NOP	
CALL SAM (A(I), B+C, EXTSUB, S)		
.		
.		

AUTOCODER SUBPROGRAM

LABEL	OPERATION CODE	OPERAND	COMMENTS
	TITLE	SAM	
	SBR	X13	
	SBR	L6+5	Place X13
	A	+21,L6+5	+21 into return instruction
	MLCA	4+X13,L1+5	Move address constants
	MLCA	9+X13,L2+5	into floating-point
	MLCA	19+X13,L3+5	and branch instruc-
	MLCA	14+X13,L4+5	tions
	MLCA	19+X13,L5+5	
	DCWS	IBINTRP	Compute the argument for EXTSUB as the
L1	FRA	0	sum of the first,
L2	FA	0	second, and
L3	FA	0	fourth arguments of SAM
	FST	T1	Store sum in T1
L4	B	0	Calling sequence for
	DCW	T1	the external sub-
	NOP		routine
	(Result of external subroutine is conventionally stored in the accumulator.)		
	DCWS	IBINTRP	
L5	FST	0	Return result to calling program as altered fourth argument
L6	B	0	Return to calling pro-
T1	DCW	=10	gram. The call to
	END		IBINTRP destroys X13 contents so B 21+X13 cannot be used

NOTE: Linkage to routines whose names come as arguments is by a branch instruction into which an address is moved during execution. A DCWS is not used here since the name EXTSUB of the routine is unknown when the subprogram is assembled.

The DCWS IBINTRP instructions are omitted if the object program is to be run on an IBM 7010 System equipped with the Floating-Point Arithmetic feature.

CHAIN is an Operating System feature that handles the creating and loading of specified phases of a multiphase program. For the FORTRAN programmer, CHAIN provides a convenient method of segmenting a program that exceeds available core storage into several parts to be executed separately, but as a part of the same job. CHAIN also enables the FORTRAN programmer to link into one job several programs that must operate on the same data and that otherwise would be executed in several consecutive jobs.

Each segment of a program run under control of the CHAIN feature is termed a *link*. One of two or more links is the *main link*; all other links are *dependent links*. The main link is the link that is loaded and executed first; it specifies the loading and executing of any, all, or none of the dependent links. All links are assigned sequential numbers to permit specifying desired dependent links. A main link is always given the number 1; dependent links are given increasingly larger numbers, 2, 3, 4, . . . 998, 999.

Three special Linkage Loader control cards, LINK, COMN, and ENTRY, are used in defining links for a CHAIN job. For a discussion of these cards, and additional information on the CHAIN feature, see the publication *IBM 1410/7010 Operating System; System Monitor*, Form C28-0319. The information about the CHAIN feature that follows pertains only to its use with FORTRAN.

Using the CHAIN feature with FORTRAN requires only a few additional statements in main programs to be used in the CHAIN environment; no additional statements are required in any subprograms.

Main Link

A main link must meet the following speeifications:

1. It must contain a FORTRAN main program.

2. It must include CALL CHAIN statements (see "Calling Dependent Links" in this section) for any dependent links desired. Any or all (or none) of the dependent links can be called any number of times and in any order.

3. It must declare, in COMMON, all COMMON variables used in the main link and/or in any dependent link.

Dependent Links

The following specifications apply to dependent links:

1. They must be either FORTRAN main programs or SUBROUTINE subprograms that have no arguments. SUBROUTINE subprograms that have arguments and FUNCTION subprograms cannot be dependent links, but they can be called from within dependent links.

2. They should not include CALL CHAIN statements. Execution of these statements in a dependent link prohibits a return to the main link and, effectively, causes the dependent link to be treated as a new main link.

3. They must declare, in COMMON, those COMMON variables used in common with the main link.

If a dependent link is a FORTRAN main program, it should be terminated with a CALL RETURN statement (see "Exiting from Main Program Links" in this section) rather than a STOP or CALL EXIT statement. Use of the CALL RETURN statement causes a return to the main link.

NOTE: A CALL RETURN statement in a main program used as a dependent link does not prohibit the user from executing that main program outside the CHAIN environment. When executed outside the CHAIN environment, the CALL RETURN statement causes a return to the Monitor, the same result that would ordinarily be achieved with a STOP or CALL EXIT statement.

If a dependent link is a SUBROUTINE subprogram, no additional requirements are imposed. The RETURN statement, required in every SUBROUTINE subprogram, causes a return to the main link. The CALL RETURN statement must not be used in a SUBROUTINE subprogram.

A return to the main link from any dependent link is made to the statement in the main link immediately following the CALL CHAIN statement that called the dependent link.

Calling Dependent Links

General Form
CALL CHAIN (i) i is the number of the dependent link to be loaded and executed. i may be a subscripted integer variable, or an integer constant, whose value may range from 2 through 999. The main link (link number 1) cannot be called.

The CALL CHAIN statement is used in a main link to call a dependent link for execution.

EXAMPLE

The following statements could be used to cause execution of dependent links 2, 3, 4, and 2 (again), in the order noted:

```
CALL CHAIN (2)
CALL CHAIN (3)
CALL CHAIN (4)
CALL CHAIN (2)
```

Exiting from Main-Program Links

General Form
CALL RETURN

The CALL RETURN statement is used only in links (main or dependent) that are FORTRAN main programs.

In a dependent link, the CALL RETURN statement causes a return to the main link at the statement immediately following the CALL CHAIN statement that called the dependent link.

In a main link, the CALL RETURN statement causes either return to the Monitor (if no dependent links have been called), or return to the statement in the main link immediately following the last CALL CHAIN statement executed (if any dependent links have been called.)

EXAMPLE

The sequence of statements below shows two CALL RETURN statements in a main link, one before any dependent link is called, another after two dependent links have been called. The first CALL RETURN statement causes a return to the Monitor. The second CALL RETURN statement causes a return to the statement (CONTINUE) following the last CALL CHAIN statement executed.

```
      .
      .
      .
CALL RETURN
      .
      .
      .
CALL CHAIN (I)
CALL CHAIN (J)
CONTINUE
      .
      .
      .
CALL RETURN
      .
      .
      .
```

Loading of Links

The main link of a CHAIN job is loaded and executed first. The main link is always resident in core storage, unless the user specifies otherwise in the second (base) operand of a LINK card.

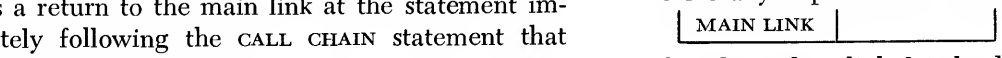
Dependent links are loaded in core storage beyond the main link. Dependent links overlay one another unless the user specifies otherwise in a LINK card.

SUBROUTINE and FUNCTION subprograms and Built-In Functions are loaded with the links that use them. If some of these subprograms are used by more than one link, the user should use the COMN card to force them to be loaded with the first link that uses them. If the COMN card is not used, the subprograms will be loaded with each link.

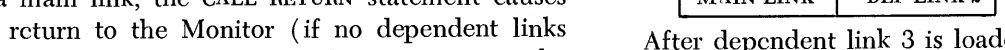
EXAMPLE

The diagrams below illustrate where dependent links 2, 3, 4 of a CHAIN job reside in core storage in relation to the main link. Assume no COMN cards were used.

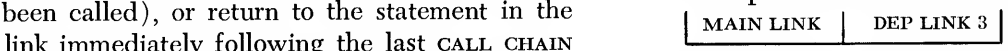
Before any dependent link is loaded:



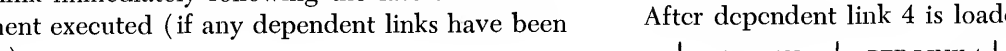
After dependent link 2 is loaded:



After dependent link 3 is loaded:



After dependent link 4 is loaded:



References Among Links

Main links may use any SUBROUTINE subprograms, FUNCTION subprograms, and/or Built-In Functions that are included in the main link. Dependent links may use any SUBROUTINE subprograms, FUNCTION subprograms, and/or Built-In Functions that are included in the dependent link itself, in the main link, or in any previously executed dependent links still in core storage. Dependent links may not use any SUBROUTINE subprograms, FUNCTION subprograms, and/or Built-In Functions included in any dependent links not yet loaded and executed.

Sample Job Using CHAIN Feature

Figure 9 shows the control cards and FORTRAN source-program decks for a job using the CHAIN feature. (If a user desires, he may punch the cards and execute the job.) The job consists of the compilation of a main link and two dependent links, the processing of the links by Linkage Loader, and the execution of the resultant program. The paragraphs below explain the job.

CONTROL CARDS

The first block of cards (MON$$ DATE through MON$$ MODE) does the following:

1. Provides the Monitor with the current date.
2. Assigns a name to the job.
3. Assigns output units required for the job.
4. Declares mode GO (because the compiled programs are to be executed after compilation).

The next three blocks of cards consist of MON$$ EXEQ FORTRAN control cards followed by source programs to be compiled.

```
      MON$$      DATE 12345
      MON$$      JOB  EXAMPLE OF A 3-LINK FORTRAN JOB USING CHAIN FEATURE.
      MON$$      ASGN MJ8,A1
      MON$$      ASGN MGO,82    GO OUT
      MON$$      MODE GO


      MON$$      EXEQ FORTRAN,,,,,,,ECRU
C     MAIN LINK (MAIN PROGRAM ECRU) COMPUTES BLOCK A OF OUTPUT.
        COMMON LISTA(4),LISTB(4),LISTC(4),LISTO(4),LISTX(4),PLIST(4),
       1AMEAN(4)
       4 FORMAT(1H1,5X,76HEXAMPLE OF A THREE-LINK JOB CONSISTING OF A MAIN
       1PROGRAM AND TWO SUBROUTINES.//////)
       5 FORMAT(15X,7HBLOCK A/)
       1 FORMAT(5X,8HLISTA    ,4I8/5X,8HLISTB    ,4I8/5X,8HLISTC    ,4I8/5X,8H
       2LISTD    ,4I8//////)
         DO 20 I=1,4
         LISTA(I)=2**I
         LISTB(I)=3**I
         LISTC(I)=4**I
         LISTO(I)=5**I
      20 CONTINUE
         WRITE(3,4)
         WRITE(3,5)
         WRITE(3,1)(LISTA(I),I=1,4),(LISTB(I),I=1,4),(LISTC(I),I=1,4),(LIST
       1O(I),I=1,4)
         I=4
         IF(LISTD(I)-1)60,70,70
      70 CALL  CHAIN(002)
      71 CALL  CHAIN(3)
      72 STOP
C     IF CALL RETURN EXECUTED, CONTROL RETURNS TO MONITOR.
      60 CALL RETURN
         END


      MON$$      EXEQ FORTRAN
        SUBROUTINE   AZURE
C     DEPENDENT LINK 2 (SUBROUTINE AZURE) COMPUTES BLOCK B OF OUTPUT.
        COMMON LISTA(4),LISTB(4),LISTC(4),LISTO(4),LISTX(4),PLIST(4),
       1AMEAN(4)
       8 FORMAT(15X,7HBLOCK A/)
       1 FORMAT(5X,8HLISTA    ,4I8/5X,8HLISTB    ,4I8/5X,8HLISTC    ,4I8/5X,
       18HLISTD    ,4I8//15X,7HBLOCK B//5X,8HLISTX    ,4I8/5X,8HAMEAN    ,
       24F8.2/////)
         DO 5 I=1,4
         LISTX(I)=LISTA(I)+LISTB(I)+LISTC(I)+LISTO(I)
         PLIST(I)=LISTX(I)
         AMEAN(I)=PLIST(I)/4.
       5 CONTINUE
         WRITE(3,8)
         WRITE(3,1)(LISTA(I),I=1,4),(LISTB(I),I=1,4),(LISTC(I),I=1,4),(LIST
       1O(I),I=1,4),(LISTX(I),I=1,4),(AMEAN(I),I=1,4)
         RETURN
         END


      MON$$      EXEQ FORTRAN
        SUBROUTINE   OCHRE
C     DEPENDENT LINK 3 (SUBROUTINE OCHRE) COMPUTES BLOCK C OF OUTPUT.
        COMMON LISTA(4),LISTB(4),LISTC(4),LISTO(4),LISTX(4),PLIST(4),
       1AMEAN(4)
      11 FORMAT(15X,7HBLOCK A,25X,7HBLOCK C/)
       1 FORMAT(5X,8HLISTA    ,5I8,F8.2/5X,8HLISTB    ,5I8,F8.2/5X,
       18HLISTC    ,5I8,F8.2/5X,8HLISTO    ,5I8,F8.2/
       2/15X,7HBLOCK B//5X,8HLISTX    ,4I8/5X,8HAMEAN    ,4F8.2/)
         DO 50 I=1,4
         LISTAT=LISTAT+LISTA(I)
         LISTBT=LISTBT+LISTB(I)
         LISTCT=LISTCT+LISTC(I)
         LISTOT=LISTOT+LISTO(I)
      50 CONTINUE
         ALIST=FLOAT(LISTAT/4)
         BLIST=FLOAT(LISTBT/4)
         CLIST=FLOAT(LISTCT/4)
         DLIST=FLOAT(LISTDT/4)
         WRITE(3,11)
         WRITE(3,1)((LISTA(I),I=1,4),LISTAT,ALIST,(LISTB(I),I=1,4),LISTBT,B
       1LIST,(LISTC(I),I=1,4),LISTCT,CLIST,(LISTD(I),I=1,4),LISTDT,DLIST,
       1(LISTX(I),I=1,4),(AMEAN(I),I=1,4))
         RETURN
         END


      MON$$      EXEQ LINKLOAD,,,,EXAMPLE2
                 LINK ECRU
                 COMN IBINTRP,FLOAT
                 LINK AZURE
                 ENTRYAZURE
                 LINK OCHRE
                 ENTRYOCHRE
                 END
      MON$$      EXEQ EXAMPLE2,MJ8
      MON$$      END
```

Figure 9. Sample Job Using CHAIN Feature

The first program compiled is FORTRAN main program ECRU; this program is the main link of the CHAIN job.

The second program compiled is FORTRAN SUBROUTINE subprogram AZURE; this subprogram is dependent link 2 of the CHAIN job.

The third program compiled is FORTRAN SUBROUTINE subprogram OCHRE; this subprogram is dependent link 3 of the CHAIN job.

The last block of cards (MON$$ EXEQ LINKLOAD through MON$$ END) does the following:

1. Causes Linkage Loader to process the three compiled programs as a CHAIN job named EXAMPLE2.

2. Defines ECRU as the main link. (The first LINK card contains the name of the main link.)

3. Causes the floating-point interpretive subroutine IBINTRP and the Built-In Function FLOAT to

be loaded before the main link. (The COMN card forces this order of loading.)

4. Defines AZURE and OCHRE as dependent links, numbered 2 and 3, respectively. (Links named in LINK cards following the first LINK card are assigned increasingly larger link numbers.)

5. Assigns entry points to dependent links 2 and 3. (An ENTRY card is required if a dependent link is other than a FORTRAN main program.)

6. Indicates the end of CHAIN control cards.

7. Causes execution of CHAIN job EXAMPLE2.

8. Defines the end of the job.

EXECUTION OF CHAIN JOB

When the CHAIN job EXAMPLE2 is executed, the following steps occur:

1. The main link ECRU is loaded (with IBINTRP and FLOAT preceding it in core storage).

2. ECRU computes and writes block A of the three blocks of output produced by EXAMPLE2.

3. ECRU calls dependent link 2 (AZURE). This is accomplished by the CALL CHAIN statement numbered 70.

4. AZURE is loaded into core storage beyond ECRU.

5. AZURE computes block B and writes blocks A and B of the output.

6. AZURE returns control to the main link ECRU at the statement numbered 71, the statement immediately following the CALL CHAIN statement that called AZURE.

7. ECRU calls dependent link 3 (OCHRE). This is accomplished by the CALL CHAIN statement numbered 71.

8. OCHRE is loaded into core storage beyond the main link; OCHRE overlays dependent link 2 (AZURE).

9. OCHRE computes block C and writes blocks A, B, and C of the output.

10. OCHRE returns control to the main link ECRU at the statement numbered 72, the statement immediately following the CALL CHAIN statement that called OCHRE.

11. ECRU returns control to the Monitor. This is accomplished by the STOP statement.

NOTE: If the CALL RETURN statement in the main link is executed, control returns to Monitor because no dependent links have been called.

PART 3 — DIAGNOSTIC AND ERROR MESSAGES

Diagnostic Messages

Diagnostic message numbers appearing on the source program listing, together with the meaning of the message number, are given below.

Compilation of the source program is not attempted (and object program execution is cancelled in a compile-and-go operation) for all diagnostic messages listed except numbers 003, 005, 020, 022, 038, and 096. These warning messages are indicated by an asterisk in the accompanying list. The asterisk is *not* printed with the message number on the source program listing.

MESSAGE NUMBER	MESSAGE
001	Statement not recognized
002	Improper statement as a relational IF trailer
003*	Blank statement text beginning with column 7
004	More than ten cards per statement
005*	One or more of following illegal character(s) found in statement: record mark, group mark, exclamation mark. The character has been deleted
006	Premature end of statement
007	Uneven parentheses count or missing parenthesis
008	Improper or invalid character
009	SUBROUTINE or FUNCTION is not first statement
010	REAL, INTEGER, or EXTERNAL follow a DIMENSION, COMMON, EQUIVALENCE, or FORMAT
011	A specification statement appears after the first executable statement
012	DIMENSION, COMMON, or EQUIVALENCE are out of prescribed order
013	An arithmetic statement function appears after an executable statement
014	A DIMENSION, COMMON, or EQUIVALENCE follows a FORMAT statement
015	The object of GO TO or Arithmetie IF is a non-executable statement
016	Improper statement for the end of a DO range
017	Object of a GO TO or Arithmetic IF statement missing
018	FORMAT statement is not prior to an I/O statement reference
019	End of DO range missing
020*	Coding which will not be executed appears after a GO TO, RETURN, STOP, or Arithmetic IF
021	Overlapping DO statements
022*	END card is missing but assumed by processor
023	Statement number is zero or has a non-numeric character in it
024	Statement number appears on more than one statement
025	Character must be numeric
026	A number in E notation cannot have more than two exponent digits
027	A number must follow the sign in E notation
028	An improper or invalid character follows number
029	A decimal point with neither leading nor trailing digits
030	First character of a name must be alphabetic
031	Name greater than six characters
032	Name must be an integer name that is not DIMENSIONed
033	A name has been declared in COMMON more than once
034	A name has been DIMENSIONed more than once
035	A name has had mode declared more than once
036	A name has been declared in EXTERNAL twice; or in EXTERNAL and DIMENSION or COMMON or EQUIVALENCE
037	RETURN statement found in a main program
038*	RETURN statement was not included in this subprogram
039	A FUNCTION statement must have at least one argument
040	Improper or invalid character in FUNCTION or SUBROUTINE statement
041	Declarative statement subscripts cannot be zero
042	Improper character after right parenthesis in DIMENSION
043	More than a three-dimensional array
044	Improper character after name in COMMON
045	Bad punctuation, or subscripting a variable that is not DIMENSIONed, in an EQUIVALENCE statement
046	More than one name required in parentheses of EQUIVALENCE
047	DIMENSIONed name in EQUIVALENCE must have one and only one integer subscript
048	Nothing should follow number in PAUSE
049	Nothing should follow the word STOP, CONTINUE, RETURN, or END
050	Neither computed nor simple GO TO
051	Computed GO TO needs a comma after the right parenthesis
052	A comma or right parenthesis should follow the number
053	Nothing should follow the name in computed GO TO, the statement number in the Unconditional GO TO, or the third statement number in an Arithmetic IF

MESSAGE NUMBER	MESSAGE
054	A GO TO or Arithmetic IF is going to itself
055	End of the DO range occurred before or at the DO statement
056	Comma after statement referenee in DO
057	An equal sign must follow index in DO
058	DO nest greater than 25
059	Nothing should follow the parameters of a DO
060	End of DO statement found after equal sign or comma of index parameters
061	A DO should have two or three parameters
062	An improper eharaeter follows unit number or name in I/O statement
063	A name that is not DIMENSIONed eannot have a subseript
064	Improper eharacter follows name, subscript, or right parenthesis
065	A right parenthesis must follow index parameters in an I/O list
066	No unit designation has been given, or its first charaeter is improper
067	Unit number should be 1-9, one digit
068	Nothing should follow unit designation in BACKSPACE, REWIND, or ENDFILE
069	The subroutine name is DIMENSIONed
070	A left parenthesis does not follow SUBROUTINE name
071	Right parenthesis of CALL is not last eharaeter in the statement
072	Parentheses eount within argument of CALL or expression of Relational IF is uneven
073	Relational IF statement has no trailer
074	Mixed mode between expressions of a Relational IF
075	Left-hand side of arithmetie statement does not have an equal sign after first variable
076	Illegal eonseeutive operators
077	Mixed mode
078	Arithmetic statement ends with an opcrator
079	Right parenthesis follows an operator
080	Integer**real not permitted
081	A**B**C not allowed
082	Comma improper
083	Character is neither arithmetic operator nor punetuation
084	Improper eharaeter follows exponent operator
085	Number of subscripts not equal to the number deelared
086	+ or − in subscript not followed by a number
087	A real number is in the subseript
088	Improper or invalid eharaeter within a subscript; or nothing within parentheses; or end of statement within subseript
089	A nonsubscripted array name appears in an arithmetie expression or IF
090	Invalid use of nonsubseripted array name in FUNCTION or CALL
091	An argument in an arithmetic statement function defining statcment is DIMENSIONed
092	An argument in an arithmetic statement funetion defining statement contains an improper or invalid eharaeter
093	Subscripted array in arithmetic statement funetion defining statement
094	No eoefficient for P, X, or H eonversion
095	Oetal conversion is not handled
096*	Missing statement number on FORMAT statement
097	More than 133-eharaeter line produeed by FORMAT
098	Nested parentheses in FORMAT are not permitted
099	Missing period in E or F within FORMAT statement
100	Number after period in E or F is missing or is greater than two digits within the FORMAT statement
101	A, I, E, or F must be followed by a number in the FORMAT statement
102	Improper charaeter follows a number in a format speeifieation or the FORMAT statement ends prematurely
103	Two eonsecutive eommas
104	More than three digits appear in a number in FORMAT
105	A number must follow the minus sign
106	Incorrect count for H conversion
107	Improper or invalid character follows right parenthesis in FORMAT
108	Two eonseeutive P's preeeding a speeifieation
109	Number table/name table entry overflow. Too many names and statement number uses
110	Too many arguments appear in arithmetie statement funetion definition statement
111	Too many names declared in COMMON
112	Too many DIMENSIONed names
113	A comma must follow nX in FORMAT
114	The number appearing in the statement is too long
115	Too many (more than 9999) eharacters appear in FORMAT statements
116	Program is too large to compile (dietionary overflow has oeeurred) or run (more than 100,000 positions of eore storage required by this program). If a single array exeeeds 100,000 positions, the overflow is not detected
117	The EQUIVALENCE speeifieation is ineonsistent
118	Two variables in COMMON have been equivalenced
119	EQUIVALENCE has extended common upward
120	A zero coefficient appears in a FORMAT specification other than P, or width in E, F, I, or A specifications is zero
121	Threc statement numbers are required in an Arithmetic IF statement
122	A comma should separate the statement numbers in the Arithmetic IF statement

*Warning message; compilation continues and objeet program execution is not canceled.

Error-message numbers are printed on the Standard Print Unit during program execution in this format: ERROR NO. XXX AT LOCATION XXXXX. The message corresponding to each error number appears in the accompanying list.

Information concerning unusual end of program appears in the publication, *IBM 1410/7010 Operating System; System Monitor,* Form C28-0319.

MESSAGE NUMBER	MESSAGE	ACTION TAKEN
805	INT built-in function. Integer larger than integer field	Next sequential instruction
806	IFIX built-in function. Integer larger than integer field	Next sequential instruction
807	AMOD built-in function. Modulus is zero (real)	Result is always zero
808	MOD built-in function. Modulus is zero (integer)	Result is always zero
809	Logarithm of negative number or zero (ALOG built-in function) or a negative number with non-integer exponent	Unusual end of program
810	SQRT built-in function. Square root of negative number	Square root of absolute value is ealculated
811	Negative exponent (integer) for expression of form I**J	Result is always integer 1
812	SLITE or SLITET built-in function. Sense light must be 0, 1, 2, 3, or 4 for SLITE and 1, 2, 3, or 4 for SLITET	Unusual end of program
815	IDIM built-in function. Arg 1-Arg 2 creates overflow	Next sequential instruction
820	EXP built-in function. Argument must be less than 225.	Unusual end of program
821	COS or SIN built-in function. Argument must be less than 10000. radians in absolute value	Unusual end of program
850	An invalid I/O command has been given; e.g., a REWIND of the Standard Input Unit	Unusual end of program
851	An I/O command addressing the Standard Input, Punch, or Print Unit has been given without a FORMAT statement	Unusual end of program
852	The FORMAT statement contains an invalid symbol	Unusual end of program
853	The data for E or F input contains an invalid symbol	Unusual end of program
854	FORMAT statement is invalid	Unusual end of program
856	Field width specified for output data is too small for I conversion	High-order digits are truncated and an * will be inserted in the most significant digit position
857	Field width specified for output data is too small for E or F conversion	An * is inserted in high-order numeric position, any other digits preceding decimal point are replaced by blanks, decimal point is inserted, and digits following decimal point are replaced by zeros. For E conversion, the correct digits are inserted in the exponent field
860	A permanent read/write error has been detected	Normal end of program
861	An end-of-file condition has occurred. (FORTRAN does not handle multireel files. The user can write his FORTRAN program to allow for multireel files by using a code or signal at the end of each reel, testing for it in his program, and switching to another recl. For this procedure, alternate units must be assigned.)	Normal end of program
862	A BACKSPACE statement has addressed a tape which has not been referenced by a READ or WRITE	BACKSPACE statement will be ignored

Appendixes

Appendix A: Source Program Statements and Sequencing

The following set of rules describes the order in which source program statements of a FORTRAN program are executed.

Control originates at the first executable statement. The Specification statements, and the FORMAT, FUNCTION, and SUBROUTINE statements, are nonexecutable. For determination of sequencing, these statements can be ignored.

If control is with statement S, then control will pass to the statement indicated by the normal sequencing of statement S (see "Table of Source Program Statement Sequencing"). However, if S is the last statement in the range of one or more DO's that are not satisfied, the normal sequencing of S is ignored, and DO sequencing occurs.

Every executable statement in a source program (except the first) must have some programmed path of control leading to it.

Table of Source Program Statement Sequencing

STATEMENT	NORMAL SEQUENCING
a = b	Next executable statement
BACKSPACE i	Next executable statement
CALL	First executable statement of called subprogram
COMMON	Nonexecutable
CONTINUE	Next executable statement
DIMENSION	Nonexecutable
DO	DO sequencing, then the next executable statement
END	Terminates program, nonexecutable
END FILE	Next executable statement
EQUIVALENCE	Nonexecutable
EXTERNAL	Nonexecutable
FORMAT	Nonexecutable
FUNCTION	Nonexecutable
GO TO n	Statement n
GO TO (n_1, n_2, . . . , n_m),i	Statement n_i
IF (t) s	Statement s or next executable statement, if relation t is true or false, respectively
IF (a) n_1, n_2, n_3	Statement n_1, n_2, or n_3, if expression a is less than, equal to, or greater than zero, respectively
INTEGER	Nonexecutable
PAUSE	Next executable statement
READ	Next executable statement
REAL	Nonexecutable
RETURN	The first statement, or part of a statement, following the reference to this program
REWIND	Next executable statement
STOP	Terminates the program
SUBROUTINE	Nonexecutable
WRITE	Next executable statement

Appendix B: Preparing, Checking, and Punching a Source Program

The statements of a FORTRAN source program are usually written on a standard *FORTRAN Coding Form*, Form X28-7327 (Figure 10). A sample FORTRAN source program is shown in Figure 11. This program selects the largest value from an array of numbers (identified by the variable name A).

Using the FORTRAN Coding Form

Columns 1–5 of the first line of a statement may contain a statement number that identifies the statement. This number must be less than 100,000. Blanks and leading zeros are ignored in these columns; for example, bbb50 is the same as b5bb0 and 5bbb0. A statement must not be numbered zero. All statement numbers must be unique. These statement numbers do not have to be in any sequence or order; for example, the first statement of a program may be given statement number 100 and the 50th statement in a source program may be given statement number 1. These statement numbers are used, for example, in DO loops to indicate the range of the DO loop, in the GO TO statement, and to refer to FORMAT statements.

A statement may be continued on as many as nine lines. Any line with a non-blank, non-zero column 6, is considered to be a continuation of the preceding line. The actual character used in column 6 does not have any significance. The first continuation card could have a 9 in column 6, the second card an A, the third a 2, and so on.

Columns 7-72 contain the actual FORTRAN statements. Blanks are ignored except in an H-field of a FORMAT statement.

Statements with a C in column 1 are not processed by the FORTRAN processor, but the statements appear in the source program listing as comments. If there is a C in column 1, columns 2-72 may be used for comments. Comment cards may not appear between continuation cards of a statement. Comment cards

may not immediately precede the FUNCTION or SUBROUTINE statements.

Columns 73-80 are not processed and may be used for identification.

The order of execution of the source statements is governed by the sequencing described in Appendix A.

Source Program Checklist

An early successful compilation of a FORTRAN source program is more likely if the coding is checked against the following list of commonly made errors.

ITEM TO CHECK	CODING ERROR
A conversion	Field width, w, exceeds the word size, k or f+2.
Arithmetic expressions	Real and integer numbers, both constants and variables, mixed in invalid combinations. Often, a real constant is written without a decimal point.
DO parameters	Subscripted integer variable or expression used as parameter.
FORMAT statements	FORMAT specifications and I/O list not compatible.
FORTRAN language	Misspelled FORTRAN-language word such as EQUIVALENCE.
H conversion	Incorrect count for n of nH.
Order of source deck	Specification statements or FORMAT statements are out of sequence.
Program flow	Statement transfers into the range of a DO.
	Unreferenced statement after a GO TO, Arithmetic IF, RETURN, or STOP.
	END statement encountered in program flow.
Statement numbers	Use of same statement number more than once.
	Absence of a referenced statement number.
Subprograms	FUNCTION or SUBROUTINE statement missing at beginning of a subprogram; RETURN statement missing; END statement missing.
Subprogram names	Name is same as a variable name used in this program.
SUBROUTINE statement arguments	Dummy arguments that are subscripted or equivalenced variables.
Subscripted variables	*Each* subscripted variable, including those in lists, does not appear in a DIMENSION statement.

Punching the Source Program

FORTRAN source program statements, prepared as described above, are punched into a standard FORTRAN source program card, Form 888157 (see Figure 12).

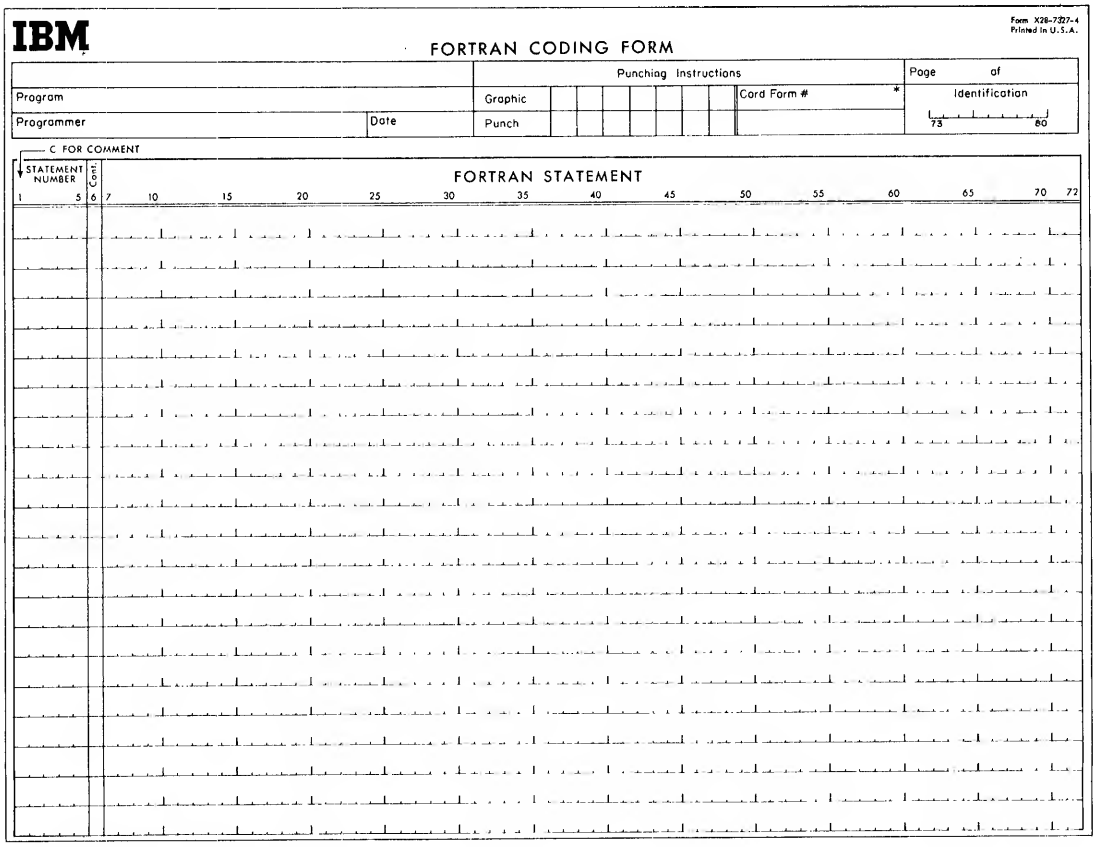
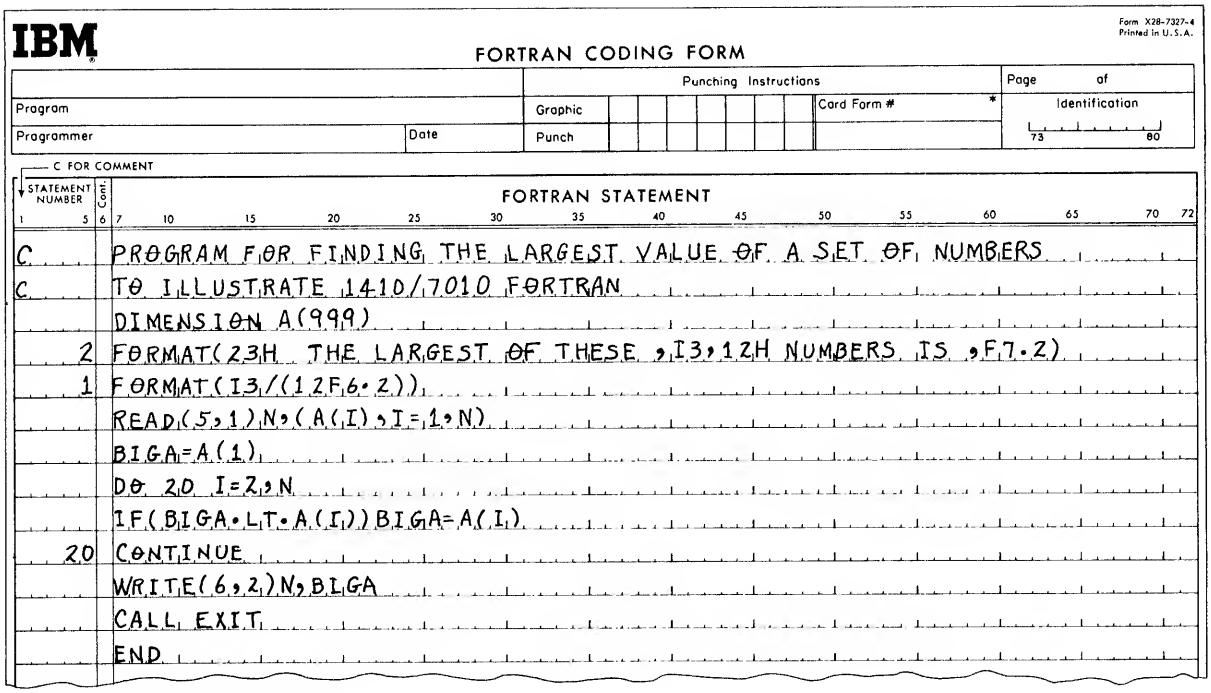


Figure 10. FORTRAN Coding Form



IBM

Form X28-7327-4
Printed in U.S.A.

FORTRAN CODING FORM

Punching Instructions

Program | Graphic | Card Form # * | Page of

Programmer | Date | Punch | Identification 73 80

C FOR COMMENT

STATEMENT NUMBER 1 5	Cont. 6	FORTRAN STATEMENT 7 10 15 20 25 30 35 40 45 50 55 60 65 70 72
C		PROGRAM FOR FINDING THE LARGEST VALUE OF A SET OF NUMBERS
C		TO ILLUSTRATE 1410/7010 FORTRAN
		DIMENSION A(999)
2		FORMAT(23H THE LARGEST OF THESE ,I3,12H NUMBERS IS ,F7.2)
1		FORMAT(I3/(12F6.2))
		READ(5,1)N,(A(I),I=1,N)
		BIGA=A(1)
		DO 20 I=2,N
		IF(BIGA.LT.A(I))BIGA=A(I)
20		CONTINUE
		WRITE(6,2)N,BIGA
		CALL EXIT
		END

Figure 11. Sample FORTRAN Source Program

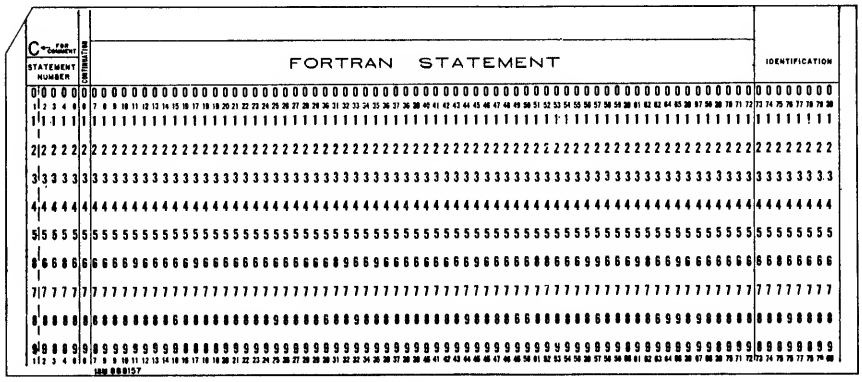


Figure 12. FORTRAN Source Program Card

Appendix C: Table of Source Program Characters

CHARACTER	CARD PUNCHES	CHARACTER	CARD PUNCHES
Blank	Blank	A	12-1
0	0	B	12-2
1	1	C	12-3
2	2	D	12-4
3	3	E	12-5
4	4	F	12-6
5	5	G	12-7
6	6	H	12-8
7	7	I	12-9
8	8	+	12
9	9	J	11-1
K	11-2	V	0-5
L	11-3	W	0-6
M	11-4	X	0-7
N	11-5	Y	0-8
O	11-6	Z	0-9
P	11-7	=	3-8
Q	11-8	'	4-8
R	11-9	.	12-3-8
-	11	)	12-4-8
/	0-1	$	11-3-8
S	0-2	*	11-4-8
T	0-3	,	0-3-8
U	0-4	(	0-4-8

NOTE: The character $ can be used in FORTRAN only in a FORMAT statement.

Index

(Where more than one page reference is given, major reference appears first.)

IBM Technical Newsletter

File Number 1410/7010-25
Re: Form No. C28-0328-3
This Newsletter No. N27-1269
Date December 30, 1966
Previous Newsletter Nos. None

IBM 1410/7010 FORTRAN

This Technical Newsletter amends the publication IBM 1410/7010 Operating System; FORTRAN, Form C28-0328-3, to include an addition concerning the SUBROUTINE subprogram.

The attached replacement pages (25-26) should be substituted for the corresponding pages now in the publication. Text changes are indicated by a vertical line to the left of the affected text.

Please file this cover letter at the back of the publication. It provides a method of determining if all changes have been received and incorporated into the publication.

IBM Corporation, Programming Publications, Dept. 637, Neighborhood Road, Kingston, N.Y. 12401

PRINTED IN U.S.A.

No card should precede the SUBROUTINE statement.

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. Any arguments so used must appear on the left side of an arithmetic statement or in an input list within the subprogram. The name of the SUBROUTINE must not be used as a variable in its SUBROUTINE subprogram.

The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. The actual arguments must correspond in number, order, and type to the dummy arguments.

When a dummy argument is an array name, a DIMENSION statement must appear in the SUBROUTINE subprogram. The corresponding actual argument in the CALL statement must also be a dimensioned array name.

None of the dummy arguments may appear in an EQUIVALENCE or COMMON statement in the SUBROUTINE subprogram.

Like the FUNCTION subprogram, the SUBROUTINE subprogram must return control to the calling program by a RETURN statement.

An END statement is also required.

Subprogram Names as Arguments — The EXTERNAL Statement

Subprogram names may be used as the actual arguments in the calling program. In order to distinguish these subprogram names from ordinary variables when they appear in an argument list, their names must appear in an EXTERNAL statement (see "The Specification Statements").

The CALL Statement

The CALL statement is used only to call a SUBROUTINE subprogram.

General Form
CALL name (a_1, a_2, . . . , a_n) name is the symbolic name of a SUBROUTINE subprogram. a_1, a_2, . . . , a_n are the actual arguments that are being supplied to the SUBROUTINE subprogram.

EXAMPLES

```
CALL MATMPY (X, 5, 40, Y, 7, 2)
CALL QDRTIC (X, Y, Z, ROOT1, ROOT2)
```

The CALL statement transfers control to the SUBROUTINE subprogram and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement. The arguments in a CALL statement may be any of the following: any type of constant, any type of subscripted or nonsubscripted variable, an arithmetic expression, the name of a subprogram.

The arguments in a CALL statement must agree in number, order, type and array size with the corresponding arguments in the SUBROUTINE subprogram.

Machine Indicator Tests

The 1410/7010 FORTRAN language provides machine indicator tests even though machine components referenced by the tests do not physically exist. The machine indicators, described below, are simulated by SUBROUTINE subprograms located in the System Library.

To use any of the following machine indicator tests, the user supplies the proper arguments and writes a CALL statement. In the following listing, i is an integer expression, j is an integer variable.

GENERAL FORM	FUNCTION
SLITE (i)	If i=0, all sense lights are turned off. If i=1, 2, 3, or 4, the corresponding sense light is turned on.
SLITET (i, j)	Sense light i (1, 2, 3, or 4) is tested and j is set to "1" or "2" if i is on or off, respectively. After the test, sense light i is turned off.
OVERFL (j)	This indicator is on if an arithmetic operation with real variables and constants results in an overflow condition; that is, if an arithmetic operation (of type real) produced a result whose value is greater than $(1-10^{-t})\times 10^{99}$. If the indicator is on, j is set to "1"; if off, j is set to "2." The indicator is set to off after the test is made.
DVCHK (j)	This indicator is set on if an arithmetic operation with real constants and variables results in the attempt to divide by zero; j is set to "1" or "2" if the indicator is on or off, respectively. The indicator is set to off after the test is made.

EXAMPLES

```
CALL SLITE (3)
CALL SLITET (K*J, L)
CALL OVERFL (J)
CALL DVCHK (I)
```

As an example of how the sense lights can be used in a program, assume that the statements CALL SLITE (1) and CALL SLITET (1, KEN) have been written. Further assume that it is desired to continue with the program if sense light 1 is on and to write results if sense light 1 is off. This can be accomplished using the Relational IF statement or a Computed GO TO statement, as follows:

```
      .
      .
IF (KEN.EQ.2) WRITE (3, 26) (ANS(K), K=1, 10)
      .
     or
      .
      .
GO TO (6, 17) KEN
17 WRITE (3, 26) (ANS(K), K=1, 10)
 6
      .
      .
```

EXIT Subroutine

A CALL to the EXIT subprogram, located in the System Library, terminates the execution of the program and returns control to the Monitor. The EXIT subprogram and the STOP statement produce identical results.

General Form
CALL EXIT

Printed in U.S.A. C28-0328-3

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601